Revised and Updated for DOS 5 & 6

Third edition

MS-DOS®

system programming

29 Pearls for DOS Programmers

edited by

David Burki
Robert Ward

DISK INCLUDED

MS-DOS System Programming

Third Edition

Edited by
David L. Burki and Robert Ward

R&D Publications
Lawrence, Kansas 66046

R&D Publications, Inc.
1601 West 23rd Street, Suite 200
Lawrence, Kansas 66046-2700
USA

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where R&D is aware of a trademark claim, the product name appears in itital capital letters, in all capital letters, or in accordance with the vendor's capitalization preference. Readers should contact the appropriate companies for more complete information on trademarks and trademark registrations. All trademarks and registered trademarks in this book are the property of their respective holders.

Copyright © 1994 by R&D Publications, Inc., except where noted otherwise. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher; with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

The programs in this book are presented for instructional value. The programs have been carefully tested, but are not guaranteed for any particular purpose. The publisher does not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher assumes no liability for damages resulting from the use of the information in this book or for any infringement of the intellectual property rights of third parties which would result form the use of this information.

Distributed by Prentice Hall
ISBN 0-13-207382-X

Cover Design: T. Watson Bogaard

Preface

“The report of my death was an exaggeration.” Mark Twain, 1897

Despite what many industry giants would have us believe, DOS is alive and well and running on millions of machines around the world. In many cases, DOS is the only operating system on these machines and will be for years to come. And as long as there are DOS machines, those machines will need DOS-based programs that run faster and do more with less.

This book is an eclectic collection of tools for the developer who is writing those lean, efficient modern-day DOS programs. You won’t find generic tutorials here, or anotated, alphabetized testimonials of every known DOS command. What you will find is a bundle of tricks for faster, cleaner code.

Some are ready-to-use modules you can drop directly into your own programs. Others are development tools designed to help you get more for your time. You’ll find detailed discussions of undocumented functions and thorough explanations of hardware interfaces. All programs have been compiled and tested under DOS v6.0 and a beta version of DR DOS v7.0. Twenty of the chapters are new to the 3rd Edition. Seven have never been published anywhere, and the others (in the tradition of the *Systems*

Programming series) are the sage advice from the experts at *Tech Specialist* and the *Windows/DOS Developer's Journal*.

Read on for the Table of Contents. Is there something here for you? If one chapter saves you one hour, you've recovered the price. And you might find much much more...

Table of Contents

DOS Tips and Tricks

Chapter 1 ***Accessing the Global MS-DOS Environment***

By Scott Robert Ladd 1

Retrieve and change the Master Environment.

Chapter 2 ***Navigating the DOS File Allocation Table***

By Robert L. Hummel 13

Access the low-level structures beneath the DOS file system.

Chapter 3 ***Modifying DOS Boots***

By Steven K. Graham 45

Everything you need to write a custom boot program.

Chapter 4 *Converting a Microsoft C Program into a TSR*

By Michaael J. Young *77*

Convert a Microsoft C program to a TSR
through a single function call.

Chapter 5 *Loading TSRs into Expanded Memory*

By Thomas W. Olsen *115*

Keep more of the lower 640Kb available
for DOS applications.

Chapter 6 *Making Windows and DOS Programs Talk*

By Thomas W. Olsen *131*

How DOS programs can share data
with Windows applications.

Chapter 7 *A Swapping Replacement for the* `spawn()` *Family*

By Ralf Brown *163*

Load and execute other applications from within your
program — using as little as 500 bytes overhead.

Chapter 8 *Exploiting the 386 Four-byte Moves*

By Davie Lee Reed *177*

Take advantage of double-word moves
available on 386 and better processors.

Chapter 9 *The* `LOADALL` *Instruction*

By Robert Collins *185*

Use the undocumneted `LOADALL` instruction to access
memory above 1Mb from real mode.

Chapter 10 *Critical Error Handling*

By David L. Burki *261*

Take full control of critical error processes (like "`drive not ready`"), and replace the DOS "`Abort, Retry, Ignore`" critical error message with a message of your own.

Chapter 11 **Ctrl-C** *and* **Ctrl-Break** *Handling under DOS*

By David L. Burki *301*

Get rid of the nasty ^*C* on your display and take full control of processing *Ctrl-C* and *Ctrl-Break*.

Chapter 12 *Handling DOS File Handle Limits*

By David L. Burki *319*

Open more than 20 files in your program and incease the system-wide file handle limit without making the user edit `config.sys`.

Chapter 13 *Preventing Lost Keystrokes*

By David L. Burki *357*

How DOS can lose keystrokes and a TSR that corrects the problem.

Chapter 14 *Gracefully Breaking Out of Infinite Loops*

By David L. Burki *369*

A great development tool.
Terminate any program gracefully.

Chapter 15 *Checking for Debug Information*

By Matt Pietrek *391*

Make sure you haven't built your production binaries with debug information.

Programming the Hardware

Chapter 16 ***Programming the PC Game Port***
By John F. Jarrett ***403***
The game port isn't just for games anymore.

Chapter 17 ***Using the Parallell Port for High-Speed File Transfer***
By David McCombs ***421***
How to move files fast through the 8-bit bandwidth of the parallel port.

Chapter 18 ***Demystifying Interrupt-Driven Serial Communications***
By Russell L. Jacobs ***451***
Everything you need to get started with serial communications.

Chapter 19 ***Introduction to the VGA Video Display Card***
By Rick Ladymon ***479***
The basics of VGA graphics annimation.

Chapter 20 ***The Math Coprocessor***
By Lee Daniel Crocker ***509***
Speed up floating point-intensive applications with these assembly language coprocessor routines.

Chapter 21 ***Interfacing to the PC Floppy Disk Controller***
By Larry Widing ***533***
How programs like FastBack control the floppy drives.

Chapter 22 Writing for the PC Speaker
By Robert D. Bybee *561*

How your program can produce high-quality voice and music through the PC speaker.

Chapter 23 Event Timing on MS-DOS PCs
By Phyllis K. Lang *583*

Beating the 55 millisecond precision for DOS event timing.

Device Drivers

Chapter 24 Writing Device Drivers with Turbo C
By Robert Allen *595*

The advantages of Turbo C for device drivers in C.

Chapter 25 Using CD-ROM Device Drivers
By Dirk S. Baum *609*

Playing music from your CD-ROM drive.

Chapter 26 DMA Revealed
By Karen Hazzah *651*

The mysterious world of Direct Memory Access (DMA) and how to use DMA in your programs.

Chapter 27 Self-Loading Device Drivers for DOS
By Thomas Nelson *687*

Build device drivers that load directly from the command line (or automatically from `autoexec.bat`).

Chapter 28 ***Interfacing a Windows Program to a Real-Mode Device Driver***

By Charles Mirho *727*

Make DOS device driver services available to protected-mode Windows applications.

Chapter 29 ***DDE to DOS***

By Guy Eddon *757*

Let a DOS program act as a DDE server for a Windows program.

Index *787*

Chapter 1

Accessing the Global MS-DOS Environment

Scott Robert Ladd

Developing software for MS-DOS can be frustrating. I often find what seem to be artificial restrictions on what can and can't be done. For example, some of the utilities provided with MS-DOS can accomplish tasks which a programmer can't duplicate by using the documented features of the operating system. This chapter lifts the veil from one of MS-DOS's most useful hidden secrets, explaining how to access and use the master copy of the MS-DOS environment.

The environment is a collection of null-terminated strings maintained by `COMMAND.COM`. Each null-terminated string is an environment variable consisting of a name and an associated text value separated by an equal sign. MS-DOS and application programs use environment variables for a wide variety of purposes. Common examples of environment variables include `COMSPEC` (which stores the path name of the MS-DOS shell), `PROMPT` (the prompt definition string), and `PATH` (containing the list of directories to be searched for executable files). While there are specific commands for changing the values of *PATH* and *PROMPT*, all other environment variables are manipulated via the MS-DOS *SET* command.

There is usually more than one copy of the environment in memory. The master environment table is created at boot time, when *COMMAND.COM* allocates a block of memory that contains the master environment variables. Additional copies of the environment are created as other programs are executed. A program that executes another program is known as a "parent," while the program it executed is called a "child." A child can be the parent of other programs. MS-DOS creates a copy of the parent environment for each child. TSRs, shell programs, and multiple copies of *COMMAND.COM* all contribute to the proliferation of copies of the environment. With the exception of the master environment, all of these environments are limited, separate, and temporary. When a local environment is created, its size is only slightly larger than that required to hold all of the existing variables in the parent environment. A copy of the environment cannot be expanded, so there's almost no room to add new variables. Changes made by the child to its copy of the environment have no effect on the parent's copy — and vice versa. When a child program terminates, its local environment vanishes, too. These facts limit the usefulness of modifying a local copy of the environment.

On the other hand, it can be useful to make changes to the master environment. A program could pass along information to other programs through master environment variables. Another application could store status information for future incarnations of itself in the master environment. A TSR can use the variables in the global environment to store information. Unfortunately, MS-DOS does not provide any documented way of accessing the master environment.

To find the master copy of the environment, we need to delve into the arcane lore of MS-DOS programming.

When an MS-DOS program is executed, a 256-byte header called the Program Segment Prefix (PSP) is created for it. This header contains important operating system data. A program can locate the copy of its local environment via a segment pointer stored at offset *44* (2c hex)within the PSP. The segment of the PSP for a program's parent is stored at PSP offset *22* (16 hex). *COMMAND.COM* is its own parent, so its parent PSP pointer contains the segment of its own PSP.

Now we have a mechanism for locating the master environment. Starting with the parent PSP pointer in the PSP of the currently executing program, we can backtrack through PSPs until we find a parent pointer that is equal to the current PSP segment. That means that we have found the first `COMMAND.COM` loaded. Using the environment pointer in `COMMAND.COM`'s PSP, we can build the address of the global environment block.

Listing 1.1 is the file `ENV.ASM`. This module uses directives for high-level language interfacing and segment declaration. As presented here, `ENV.ASM` will assemble with an small model interface to `C`. Two public functions, `GlobalEnvSeg` and `GlobalEnvSize`, are defined. `GlobalEnvSeg` uses the previous algorithm to locate the master copy of the environment; it returns a 16-bit value containing the segment of the global environment.

`GlobalEnvSize` uses another undocumented feature of MS-DOS to obtain the size of the environment block. Every block of memory allocated by MS-DOS is prefaced by a 16-byte paragraph-aligned header called the Memory Control Block (MCB). The MCB contains three pieces of information: a status indicator, the segment of the owning program's PSP, and the length of the block. The status indicator is a byte which contains either the character `M` (indicating that it is a member of the MCB chain) or a `Z` (denoting this as the last MCB in the chain). The length of the block is stored as a number of 16-byte paragraphs. (For those who are curious, the characters `M` and `Z` are the initials of one of the original developers of MS-DOS.) `GlobalEnvSize` looks in the MCB for the environment block to find out how much memory has been allocated.

Listing 1.3 is a header file named `GLOBENV.H` that provides prototypes for the functions defined in `GLOBENV.C` (Listing 1.2). Three functions, `GlobalEnvGet`, `GlobalEnvPut`, and `GlobalEnvDel` respectively, retrieve, store, and delete variables in the master environment. Each of these functions checks a flag to see if initialization has taken place; if not, the `GlobalEnvSeg` and `GlobalEnvSize` functions in `ENV.ASM` are called. `GlobalEnvGet` copies the values of the environment variable specified in the parameter `NAME` into the character array, `TEXT`. Up to `n` characters of the environment variable will be copied into `TEXT`. A `1` is returned if `NAME` was found; otherwise, `GlobalEnvGet` returns `0`.

Deleting a variable is accomplished by calling `GlobalEnvDel`, which is passed the name of the variable to be deleted. Caution should be taken not to delete important MS-DOS environment variables such as `PATH` or `COMSPEC`. A cautious programmer might add checks in `GlobalEnvDel` to prevent critical variables from being deleted. A `0` is returned to indicate success; a return of `1` reflects a failure to find the variable.

`GlobalEnvPut` is called to store a new variable. It accepts the name and value of the variable as parameters. It then calls `GlobalEnvDel` to delete any current variable of the same name; and it appends the new variable information to the end of the environment. The name of the variable is converted to uppercase, as this is the convention used by the internal MS-DOS commands such as `PATH` and `SET`. If the new variable won't fit within the available space in the environment, `GlobalEnvPut` returns `0` to indicate failure. Also, if the variable will be longer than 256 bytes, a `0` is returned. This is because the MS-DOS `SET` command cannot manipulate environment variables which are longer than 256 bytes. A return value of `1` means that `GlobalEnvPut` succeeded.

`GLOBENV.C` is written in ANSI C and has been tested with Zortech C v2.18, Microsoft C v6.00A, Microsoft Quick C v2.5, and Borland Turbo C++ v1.01. It should compile and work under MS-DOS C compilers which support far pointers and software interrupt calls. As with all functions that use undocumented features of MS- DOS, it is important to realize that these functions may not work with future (or past) versions of MS-DOS.

Listing 1.1 env.asm

```
;#######################################################################
;
;   Module Identification
;     Env  v2.00
;
;   Purpose
;     Allows access to the global MS-DOS environment.
;
;   Environment
;     Language:   Intel 80x86 Assembler
;
;     Assemblers: Microsoft Macro Assembler v5.10
;                 Borland Turbo Assembler v1.1
;
;   Requirements
;     Hardware:   IBM PC compatible
;     Software:   MS-DOS 2.11 thru 3.30, and 4.01
;                 Does not work with some buggy versions of MS-DOS 4.00
;
;   Author Information
;     Written by: Scott Robert Ladd
;
;   Legal Stuff
;     This module has been placed in the public domain.
;
;#######################################################################

TITLE   env.asm
NAME    env

.MODEL SMALL,C

.DATA

EnvSeg DW 0

.CODE

PUBLIC GlobalEnvSeg
PUBLIC GlobalEnvSize

GlobalEnvSeg PROC

        mov     ah, 62h         ; get PSP segment of current program
        int     21h
```

Listing 1.1 *continued*

```
    findshell:

        mov     es, bx              ; get address of parent program

        mov     ax, es:[22]
        cmp     ax, bx              ; if parent PSP = this PSP, shell
        je      found

        mov     bx, ax

        jmp     findshell

    found:

        mov     ax, es:[44]         ; load AX with env seg in shell PSP
        or      ax, ax              ; check for 0
        jnz     env_ptr

        mov     ax, es              ; mov ES to AX where it can be changed
        dec     ax                  ; dec AX to move back one paragraph
        mov     es, ax              ; mov new segment to ES
        mov     bx, 3               ; set BX to point to size of shell MCB
        add     ax, es:[bx]         ; add size of shell MCB to ax
        inc     ax                  ; increment AX so it points to env MCB

    env_ptr:

        mov     EnvSeg, ax          ; save environment segment
        ret                         ; return

GlobalEnvSeg ENDP

GlobalEnvSize PROC

        mov     ax, EnvSeg
        or      ax, ax
        jz      no_init

        dec     ax
        mov     es, ax
        mov     ax, es:[3]
        mov     cl,4
        shl     ax,cl

    no_init:

        ret

GlobalEnvSize ENDP

END
```

Listing 1.2 `globenv.c`

```
/*
    GLOBAL ENVIRONMENT ACCESS
        globenv.c  13-Dec-1990

        Written by Scott Robert Ladd.

*/

#include "dos.h"      /* _PSP definition */
#include "stddef.h"   /* NULL and other standard definitions */
#include "string.h"   /* string functions */
#include "globenv.h" /* to get prototypes */

/* prototypes for assembly language functions */

unsigned int GlobalEnvSeg(void);
unsigned int GlobalEnvSize(void);

/* define a macro for MSC and QC */
#if !defined(MK_FP)
    #define MK_FP(seg,off) ((void far *) (((long)(seg)<<16L)|(long)(off)))
#endif

/* global declarations */

static char far * EnvPtr;     /* pointer to first byte of global env. */
static unsigned   EnvLen;     /* BlockLength of global environment */
static int        Inited = 0; /* indicates if module is Inited */

#define NUL '\x00'

int GlobalEnvGet(const char * name, char * text, size_t n)
    {
    char far * ePtr;
    const char * nPtr;
    size_t i;

    /* check for initialization */

    if (!Inited)
        {
        EnvPtr = MK_FP(GlobalEnvSeg(),0);
        EnvLen = GlobalEnvSize();
        }
```

Listing 1.2 *continued*

```
ePtr = EnvPtr;

/* search for name */

while (*ePtr)
            {
    nPtr = name;

    while ((*ePtr == *nPtr) && (*ePtr != '=') && (*ePtr != NUL) && (*nPtr != NUL))
        {
        ++ePtr;
        ++nPtr;
        }

    if ((*nPtr == NUL) && (*ePtr == '=')) /* name found */
         {
         /* copy variable value to s */

         ++ePtr;

         /* strcpy cannot be used since pointer sizes may differ */

         for (i = 0; (i < n) && (*ePtr != NUL); ++i)
              {
              text[i] = *ePtr;
              ++ePtr;
              }

         text[i] = NUL;

         return 1;
         }

    /* skip to the next environment variable */
    while (*ePtr != NUL)
        ++ePtr;

    ++ePtr;
    }

/* name wasn't found */
return 0;
}
```

Listing 1.2 *continued*

```
int GlobalEnvPut(const char * name, const char * text)
    {
    char far * ePtr;
    size_t     l;
    char *     tPtr;
    static char temp[256];

    /* check for initialization */

    if (!Inited)
        {
        EnvPtr = MK_FP(GlobalEnvSeg(),0);
        EnvLen = GlobalEnvSize();
        }

    /* check to see that the storage required is < 256 bytes */

    if ((strlen(name) + 2 + strlen(text)) > 256)
        return 1;

    /* make a complete environment string from the components given */

    strcpy(temp,name);
    strupr(temp);
    strcat(temp,"=");
    strcat(temp,text);

    /* delete any existing variables of the same name */

    GlobalEnvDel(name);

    /* find the end of the current variables (mark by two nulls) */

    ePtr = EnvPtr;
    l = 0;

    /* loop until two NULs are found, signifying the end of the */
    /* environment */

    while (!((*ePtr == NUL) && (*(ePtr+1) == NUL)))
        {
        ++ePtr;
        ++l;
        }
```

Listing 1.2 *continued*

```
    /* get the amount of remaining space */

    l = EnvLen - l - 3;

    /* if the new variable won't fit, return an error */

    if (l <= strlen(temp))
         return 1;

    tPtr = temp;

    /* otherwise, copy s onto the end of the current environment */

    ++ePtr;

    while (*tPtr != NUL)
        *ePtr++ = *tPtr++;

    /* end the environment with two NUL bytes */

    *ePtr = NUL;
    *(ePtr + 1) = NUL;

    /* it worked! */

    return 0;
    }

int GlobalEnvDel(const char * name)
    {
    char far *   ePtr1;  /* marks beginning of next variable */
    char far *   ePtr2;  /* marks beginning of the variable */
    const char * nPtr;   /* name pointer used in search */

    int searching = 1;   /* flag to indicate search end */

    /* has the module been Inited? If not, do it! */

    if (!Inited)
        {
        EnvPtr = MK_FP(GlobalEnvSeg(),0);
        EnvLen = GlobalEnvSize();
        }
```

Listing 1.2 *continued*

```
    ePtr1 = EnvPtr;

    /* find the beginning of the variable to be deleted */

    while ((*ePtr1 != NUL) && (searching))
        {
        nPtr  = name;
        ePtr2 = ePtr1;

        while ((*ePtr1 == *nPtr) && (*ePtr1 != '=') && (*ePtr1 != NUL) && (*nPtr  != NUL))
            {
            ++ePtr1;
            ++nPtr;
            }

        if ((!*nPtr) && (*ePtr1 == '='))
            searching = 0; /* the variable we want was found! */

        while (*ePtr1 != NUL)
            ++ePtr1;

        ++ePtr1;
        }

    /* if name wasn't found, return with an error */

    if ((*ePtr1 == NUL) && (searching))
      return 1;

    /* otherwise, copy the remainder of the environment over name */

    if (*ePtr1)
        {
        while (!((*ePtr1 == NUL) && (*(ePtr1+1) == NUL)))
            {
            *ePtr2 = *ePtr1;
            ePtr2++;
            ePtr1++;
            }
        }

    /* end the environment with double NUL bytes */

    *ePtr2 = NUL;
    *(ePtr2 + 1) = NUL;

    /* it worked */

    return 0;
    }

/* End of File */
```

Listing 1.3 `globenv.h`

```
/*
    GLOBAL ENVIRONMENT ACCESS
        globenv.h   13-Dec-1990

        Written by Scott Robert Ladd.

*/

int GlobalEnvGet(const char * name, char * text, size_t n);
int GlobalEnvPut(const char * name, const char * text);
int GlobalEnvDel(const char * name);

/* End of File */
```

Chapter 2

Navigating the DOS File Allocation Table

Robert L. Hummel

The DOS file system, in use on more than 100 million PCs, is little changed from its ancestral form that appeared over a decade ago in DOS v1.0. Although it lacks some amenities, such as long filenames, the DOS file system has proven its mettle.

DOS provides your application with a set of simple services for manipulating files. Using these services, the application can open and close files, read and write data, and provide sequential and pseudo-random access to the data in those files. Beyond this file-based model, however, DOS provides no direct access to the low-level structures that make up its file system. Your program can't use DOS functions to determine directly how a file is stored on a drive or examine the subdirectory structures. To examine these aspects of the DOS file system requires a detailed knowledge of a particular low-level drive structure called the File Allocation Table (FAT).

Of all the structures used in DOS, the FAT is probably both the most well-known and least understood. The FAT underpins the DOS file system, holding the information that describes where files are stored and which sectors of the drive cannot be used for storage. Because of this, a thorough understanding of the FAT is mandatory for writing most advanced programs such as disk organizers and file undelete utilities.

Despite the lack of direct access, it isn't difficult for your program to access and analyze the FAT. To demonstrate accessing and interpreting the FAT, I present *FATWALK*, a simple program that analyzes how a file is stored on a drive by tracing through the FAT. The utility identifies the clusters holding the file's data and determines the degree of file fragmentation. Before diving into the details, however, I'll present a brief review of how a DOS drive is organized.

DOS Drive Organization

Before DOS can use a disk for storage, a logical structure must be imposed onto the disk's physical structure. The sectors of the drive are divided into four areas, each of which has a unique purpose. The four areas are: the boot record, the file allocation table (FAT), the root directory, and the data area. The size of each of these sections is determined by the drive's storage capacity, but each area must be present for a drive to be recognized by DOS.

The boot record is usually a single sector in length and always starts at the first logical sector (sector 0) of the drive. The boot record contains a short bootstrap program that the BIOS loads when attempting to boot from that drive. The bootstrap program will either load the DOS system files from the drive or display a diagnostic message indicating that the files aren't present.

In addition to the bootstrap program, the boot record contains a data area, called the BIOS Parameter Block (BPB), that delineates the physical parameters of the drive. The device drivers built into DOS use this information to determine how to perform the translation between the drive's logical structure and the underlying disk's physical structure.

The FAT appears immediately following the boot record. The FAT is the core of the DOS file system, and its contents describe how the drive's data sectors are allocated. Each entry in the FAT corresponds to a unique area of the drive, indicating if that area is free, in-use, or unusable (typically due to physical defects in the media). In most cases, DOS creates and maintains two identical copies of the FAT.

The root directory is a fixed-length structure that contains a list of directory entries. Each entry contains a file's name, attributes, time and date of creation, time and date of the most recent modification, and a pointer to the first group of data sectors used by the file.

Programs, program data, and subdirectory structures are stored in the remainder of the disk's sectors known as the data area. Data sectors are allocated to files as required. In general, allocating and tracking individual data sectors on a large disk would require so much overhead that file handling efficiency would plummet. To improve the drive's performance, data sectors are generally allocated in groups called clusters. The number of sectors in a cluster is dependent on the type of the disk. A 1.44Mb floppy, for example, has one sector per cluster. On a hard disk, permissible cluster sizes fall within the range from four to 64 sectors per cluster, depending on your DOS version. Regardless of the type of disk, the number of sectors per cluster will always be an integral power of 2.

FAT Mechanics

The FAT is DOS's drive space allocation and control mechanism. Simply put, the FAT is a singly-linked list of cluster numbers that also serves as an index to the drive sectors used to store each file or subdirectory on the drive. The head of the linked list (the number of the first cluster that belongs to a file) is stored as part of that file's directory entry. The value stored in the FAT at the index value specified by the first cluster number contains the second cluster number. That cluster contains the number of the subsequent cluster, and so on. A reserved value is used to indicate that any particular cluster is the last cluster in a list. By tracing this list through the FAT, the number of each cluster belonging to the file (and therefore all sectors belonging to the file) can be found.

The specific value stored in a FAT entry indicates whether the corresponding area of the drive is unused and available, in-use, or unusable (typically due to defects on the disk surface). DOS constructs a FAT using either 12-bit or 16-bit entries, depending on the capacity of the drive. In all versions of DOS from v2.0 to v6.3, if the number of clusters reported by the drive is less than 4086 clusters, the FAT is created with 12-bit entries. If the drive reports 4086 or more clusters, 16-bit FAT entries are used. (Non-DOS formatting programs may use different criteria.)

Decoding the FAT

Decoding the 12-bit FAT

First, obtain the starting cluster number from the directory entry. To locate subsequent clusters, perform the following steps.

1. Multiply the cluster number by 1.5 (the size of a FAT entry in bytes). This is easily done by first multiplying the value by 3 then dividing by 2.
2. Take the whole part of the product as a byte offset into the FAT. For example, multiplying *235h* * 1.5 gives a whole product of *34Fh*.
3. Move the word at the calculated offset into a register. The offset is relative to the beginning of the FAT, so you must determine the sector in which the entry is located and read that sector into memory. (Note that because a FAT entry is 1.5 bytes long, it may span a sector boundary.)
4. If the cluster number used in step 1 was an even number (its low-order bit was 0), keep the low-order 12 bits of the word retrieved in step 3 by *AND*ing the word with *FFFh*. Otherwise, keep the high-order 12 bits by shifting the word right 4 bits.

Because of the two possible FAT types, a unique notation is used to express FAT values. Each FAT entry is represented by a three-digit (for a 12-bit FAT) or four-digit (for a 16-bit FAT) hexadecimal number. When a FAT value is shown as *`(x)yyy`*, the digit *`x`* represents the high-order 4 bits and is used only for 16-bit FAT entries.

5. If the resultant value is in the range *`FF8h`* to *`FFFh`*, no more clusters are in the file. Otherwise, the value represents the number of the next cluster in the file.

Decoding the 16-bit FAT

First, obtain the starting cluster number from the directory entry. To locate subsequent clusters, perform the following steps.

1. Multiply the cluster number by 2.
2. Move the word at the calculated offset into a register. The offset is relative to the beginning of the FAT, so you must determine the sector in which the entry is located and read that sector into memory.
3. If the resultant value is in the range *`FFF8h`* to *`FFFFh`*, no more clusters are in the file. Otherwise, the value represents the number of the next cluster in the file.

Translating Cluster Numbers

To convert a cluster number to the first logical sector number in the cluster, use the following formula:

```
(CN - 2) * SPC + FDS
```

where *`CN`* is the cluster number, *`SPC`* is the number of sectors per cluster, and *`FDS`* is the sector number of the first data sector on the drive.

The first two FAT entries, *(0)000* and *(0)001*, are not used to track disk use. Instead, they hold a value called the "media ID byte," originally intended to identify the format of a diskette. The same ID value has been used for several different diskette formats, and all hard drives are identified as *F8h*, so this byte is essentially useless from a programming point of view. Other special FAT values are as follows:

(F)FF0h-(F)FF6h	If not part of an allocation chain, indicates reserved clusters. These values are rarely, if ever, used to mark reserved clusters.
(F)FF7h	If not part of an allocation chain, indicates that at least one of the sectors in the cluster is bad. The entire cluster is therefore unusable and programs should not attempt to read or write the indicated sectors.
(F)FF8h-(F)FFFh	This cluster is the last cluster in the allocation chain. In general, only the value *(F)FFFh* is used. The other values are rarely, if ever, seen.

Tracing a File

Every journey begins with a single step, and tracing an allocation chain through the FAT is no different. To trace a file, you must first determine the number of the first cluster allocated to the file. To get this information, you must use one of DOS's older file control block-based functions, *Find First Matching File With FCB (INT 21h, AH=11h)*.

The *FCB Find First* function searches the current directory on the current drive for the first directory entry that matches a filename. (Figure 2.1 details the fields present in an *FCB*.) If a match is found, the function returns with *AL=0* and copies the directory entry for that file to a buffer specified by the current Disk Transfer Address (DTA). The layout of a directory entry is shown in Figure 2.2. Note that the directory entry includes the number of the file's first cluster.

The *FCB Find First* function has another unique and useful property. You must use this function to open up the hidden files that DOS uses to contain subdirectory structures. Simply make the desired drive and directory the default, load the *FCB* with the "." string, indicating the current directory, and invoke the function. If the *Find First* function

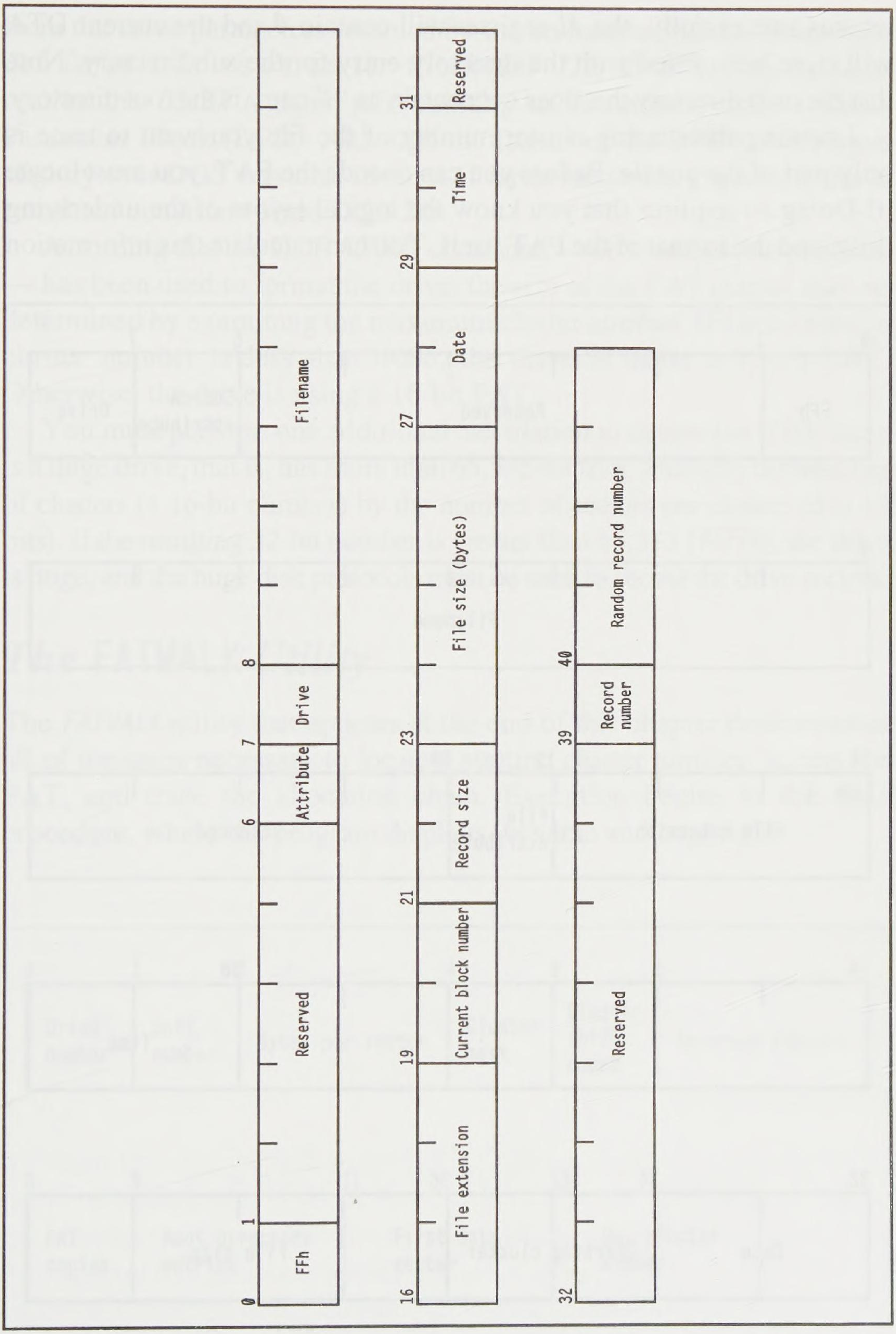

0
FFh
1
Reserved
6
Attribute
7
Drive
8
Filename
16
File extension
19
Current block number
21
Record size
23
File size (bytes)
27
Date
29
Time
31
Reserved
32
Reserved
39
Record number
40
Random record number

Figure 2.1 ***Extended File Control Block Structure.***

returns successfully, the `AL` register will contain `0` and the current DTA will have been filled with the directory entry for the subdirectory. Note that the only directory that does not contain an "." entry is the root directory.

Locating the starting cluster number of the file you want to trace is only part of the puzzle. Before you can decode the FAT, you must locate it! Doing so requires that you know the logical layout of the underlying drive and the format of the FAT itself. You can calculate this information

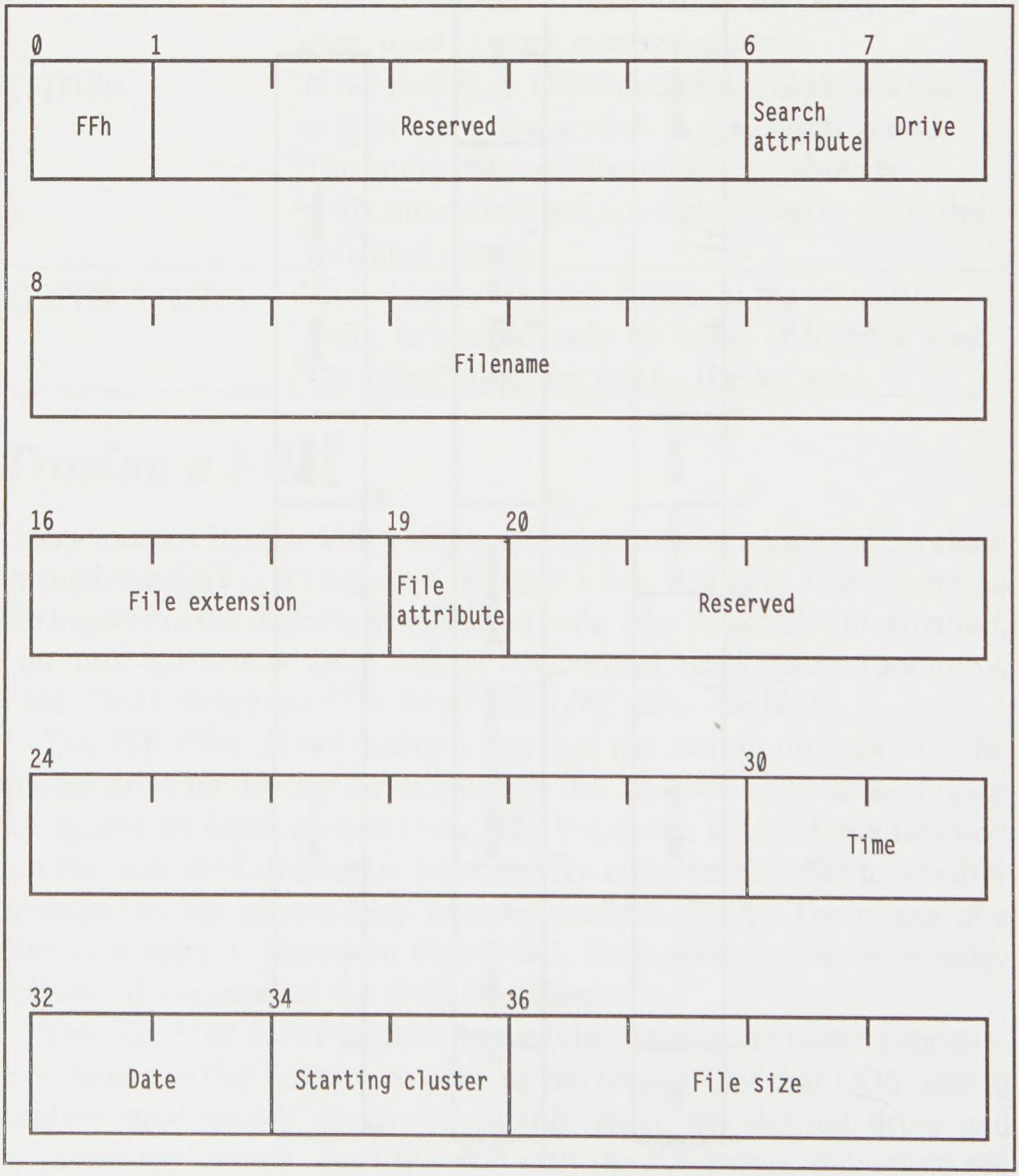

0
1
6
7
FFh
Reserved
Search attribute
Drive
8
Filename
16
19
20
File extension
File attribute
Reserved
24
30
Time
32
34
36
Date
Starting cluster
File size

Figure 2.2 *Directory Entry Structure.*

from the drive parameters returned by the undocumented DOS function `Get Default Drive Parameter Block` (`INT 21h, AH=1Fh`). This function returns the `DS:BX` register pair pointing to the current drive's Drive Parameter Block (DPB). Although the format of the DPB has changed slightly over DOS versions, the first 15 bytes (shown in Figure 2.3) have remained consistent since DOS 2.0.

Assuming that the DOS `FORMAT` command — or a compatible program — has been used to format the drive, the size of the FAT entries may be determined by examining the maximum cluster number. If the maximum cluster number is less than 4086, the drive is using a 12-bit FAT. Otherwise, the drive is using a 16-bit FAT.

You must perform one additional calculation to determine if the drive is a huge drive, that is, has more than 65,535 sectors. Multiply the number of clusters (a 16-bit number) by the number of sectors per cluster (also 16 bits). If the resulting 32-bit number is greater than 65,535 (`FFFFh`), the drive is huge, and the huge disk protocols must be used to access the drive sectors.

The FATWALK *Utility*

The `FATWALK` utility that appears at the end of this chapter demonstrates all of the steps necessary to locate a starting cluster number, access the FAT, and trace the allocation chain. Execution begins in the `MAIN` procedure, where the program displays its name and copyright.

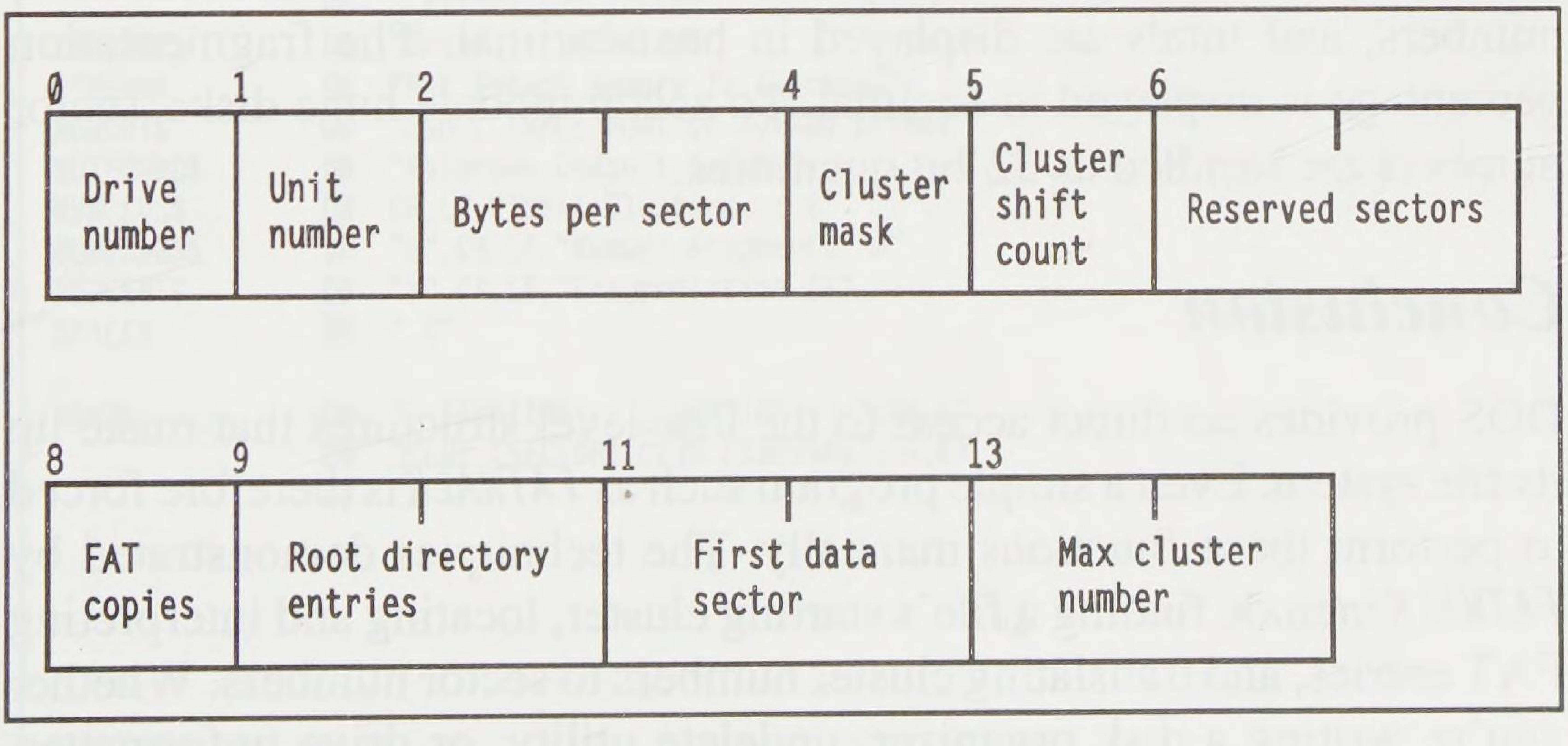

0
1
2
4
5
6
Drive number
Unit number
Bytes per sector
Cluster mask
Cluster shift count
Reserved sectors
8
9
11
13
FAT copies
Root directory entries
First data sector
Max cluster number

Figure 2.3 ***DOS Drive Parameter Block.***

The *FCB* functions predate the introduction of the DOS subdirectory structure. Because of this, *FATWALK*, must change to the drive and directory on which the file is located before calling the DOS *Find First* function. A good deal of code, therefore, is dedicated to parsing the command tail, switching to the indicated drive and directory, and preparing the file argument. After a successful call to *Find First*, the starting cluster number is retrieved from the *FCB*.

Tracing the allocation chain requires that *FATWALK* read the FAT sectors from the drive directly and interpret them — there are no DOS functions to provide this service. Most of the low-level drive information required to do so is retrieved from the DPB, retrieved using an undocumented DOS function. Other relevant drive parameters are calculated from these values, including the size of the FAT entries and whether the drive uses the huge drive protocol.

With the starting cluster number and the drive parameters in hand, walking the allocation chain is simply a matter of repeatedly calling the *VERIFY_CLUSTER*, *SHOW_CLUSTER*, and *NEXT_CLUSTER* procedures until either all the clusters in the file have been displayed or an error in the allocation chain is detected. During the tracing process, a running total is kept of the number of clusters (*NCLUS*) and the number of discontiguous fragments. The starting and ending cluster and sector numbers for each fragment are displayed during the trace.

After the allocate chain has been traced, and the cluster and sector information displayed, the summary statistics for the file or subdirectory are displayed. To keep the display code simple, all cluster numbers, sector numbers, and totals are displayed in hexadecimal. The fragmentation percentage is displayed in decimal. To accommodate huge disks, sector numbers are handled as 32-bit quantities.

Conclusion

DOS provides no direct access to the low-level structures that make up its file system. Even a simple program such as *FATWALK* is therefore forced to perform these functions manually. The techniques demonstrated by *FATWALK* include finding a file's starting cluster, locating and interpreting FAT entries, and translating cluster numbers to sector numbers. Whether you're writing a disk organizer, undelete utility, or drive unformatter, you'll find these techniques essential.

Listing 2.1 FATWALK.ASM

```
;=======================================================
; FATWALK 1.0 * Copyright (c) 1992, Robert L. Hummel
;
; FATWALK tracks a file or subdirectory's allocation
; chain through a drive's FAT. It displays the starting
; and ending cluster and sector numbers in each
; fragment and a summary of the file's fragmentation.
;-------------------------------------------------------
CSEG            SEGMENT PARA    PUBLIC  'CODE'
        ASSUME  CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG
                ORG     100H
ENTPT:          JMP     MAIN

;=======================================================
; Messages.
;-------------------------------------------------------
CR              EQU     13
LF              EQU     10
BLANK           EQU     32

COPYRIGHT$      DB  CR,LF,"FATWALK 1.0 ",254
                DB  " Copyright (c) 1992, Robert "
                DB  " L. Hummel",LF
CRLF$           DB  CR,LF,"$"

USAGE$          DB  "Usage: FATWALK pathname",CR,LF
                DB  " where pathname = [d:][path]"
                DB  "[filename.ext]",CR,LF,"$"
BADCLUS$        DB  "Cluster In Allocation Chain Was"
                DB  " Marked Bad$"
DRIVEERR$       DB  "Error Accessing Target Drive$"
FATERR$         DB  "Error Accessing Target FAT$"
INVCLUS$        DB  "Invalid Cluster Number In"
                DB  " Allocation Chain$"
INVDRV$         DB  "Drive is invalid$"
MEMERR$         DB  "Not Enough Memory To Execute$"
NOROOT$         DB  "Can't TRACE Root Or JOINed Drive$"
NOTFOUND$       DB  "Pathname Doesn't Exist$"
NUMCLUS$        DB  CR,LF,"Total Clusters  : $"
NUMFRAGS$       DB  "h",CR,LF,"Number Fragments: $"
PERCENT$        DB  "h",CR,LF,"Fragmentation %:"
SPACE$          DB  " $"

HEAD$           DB  "--STARTING--- ---ENDING----",CR,LF
                DB  "CLUS (SECTOR) CLUS (SECTOR)",CR,LF
                DB  "$"
```

Listing 2.1 *continued*

```
;------------------------------------------------------
; Program variables.
;------------------------------------------------------
FILESPEC        DW  0           ;Pointer to filespec
OLDDRV          DB  0           ;Pre-execute drive
TGTDRV          DB  0           ;Target drive
OLDDIR          DB  "\",64 DUP(0) ;Def dir on tgt drv

FES_12          EQU 00FH
FES_16          EQU 0FFH
FES             DB  FES_16      ;Assume 16-bit FAT

HUGEDISK        DW  0           ;Non-zero if huge disk
NCLUS           DW  1
NFRAG           DW  1
FATSECT         DW  -1          ;# of FAT sect in mem
;------------------------------------------------------
; Packet used for huge drives.
;------------------------------------------------------
DISK_PACKET     LABEL   BYTE

FIRST_SECTOR    DW      0,0            ;32-bit sect num
                DW      1              ;Number sects
                DW      OFFSET FATBUF  ;Buffer offset
BUF_SEG         DW      0              ; and segment
;------------------------------------------------------
; Drive data structure. Holds info about the disk
; retrieved from the Drive Parameter Block.
;------------------------------------------------------
DRIVE_DATA      LABEL   BYTE
BPS             DW  0           ;Bytes per sector
SPC             DW  0           ;Sectors per cluster
FFS             DW  0           ;First FAT sector
FDS             DW  0           ;First Data sector
MCN             DW  0           ;Maximum cluster number
;------------------------------------------------------
; This FCB is used to open the directory entry.
;------------------------------------------------------
XFCB    LABEL   BYTE            ;Extended FCB structure
        DB      0FFH            ;Extended FCB signature
        DB      5 DUP (0)       ;(Reserved)
        DB      16H             ;Search attribute
        DB      0               ;Default drive
        DB      ".",7 DUP(BLANK) ;Current dir's name
        DB      3 DUP(BLANK)    ; and extension
        DW      0               ;Current block #
        DW      0               ;Record size
        DD      0               ;Size of file (bytes)
        DW      0               ;Date
        DW      0               ;Time
        DB      8 DUP(0)        ;(Reserved)
        DB      0               ;Current record #
        DD      0               ;Random record #
```

Listing 2.1 *continued*

```
;------------------------------------------------------
; The extended directory entry structure written to the
; DTA by Find First.
;------------------------------------------------------
XDIR    LABEL   BYTE            ;Extended directory
        DB      0FFH            ;Extended FCB signature
        DB      5 DUP (0)       ;(Reserved)
        DB      0               ;Search attribute used
        DB      0               ;Drive
        DB      8 DUP (0)       ;Name
        DB      3 DUP (0)       ;Extension
        DB      0               ;File's attribute
        DB      10 DUP (0)      ;Reserved
        DW      0               ;Time
        DW      0               ;Date
FCLUS   DW      0               ;Starting cluster
        DD      0               ;File's size

;======================================================
; MAIN procedure.
;------------------------------------------------------
MAIN            PROC    NEAR
        ASSUME  CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG
;------------------------------------------------------
; Perform normal initialization and display the program
; name.
;------------------------------------------------------
        PUSH    AX              ;Save drive status

        MOV     AH,9                    ;Display
        MOV     DX,OFFSET COPYRIGHT$    ; program name
        INT     21H                     ; thru DOS

        POP     AX              ;Retrieve drive status
;------------------------------------------------------
; Change to the target drive. If a problem is found,
; returns CY and DS:DX points to an error message.
;------------------------------------------------------
        CALL    CHANGE_DRIVE    ;Change to target
        JNC     M_1
M_ERR:
        MOV     AH,9            ;Display string
        INT     21H             ; thru DOS
M_EXIT:
        MOV     AH,4CH          ;Terminate program
        INT     21H             ; Thru DOS
```

Listing 2.1 *continued*

```
;------------------------------------------------------
; Parse the command line to isolate the pathspec. A
; problem returns CY with DS:DX -> error message.
;------------------------------------------------------
M_1:
        CALL    PARSE_CMDLINE   ;Returns SI,DI
        JC      M_ERR

        MOV     [FILESPEC],SI   ;Save pointer to spec
;------------------------------------------------------
; Save the current directory on the target drive so we
; can restore it when we're done.
;------------------------------------------------------
        MOV     AH,47H                  ;Get dir of
        SUB     DL,DL                   ; current drive
        MOV     SI,OFFSET OLDDIR+1      ; store here
        INT     21H                     ; thru DOS
        JNC     M_2

        CALL    RESTORE_DRIVE           ;Restore drive
        MOV     DX,OFFSET DRIVEERR$     ;CD problem
        JMP     M_ERR
M_2:
;------------------------------------------------------
; Set the default DTA to point to our XDIR structure.
; This area is used by the find first function.
;------------------------------------------------------
        MOV     AH,1AH          ;Set DTA
        MOV     DX,OFFSET XDIR  ; to this area
        INT     21H             ; Thru DOS
;------------------------------------------------------
; Assume that the command line argument specified a
; directory name. Try to change to the new directory.
;------------------------------------------------------
        MOV     AH,3BH          ;Set directory
        MOV     DX,[FILESPEC]   ; to this string
        INT     21H             ; Thru DOS
        JC      M_3A
;------------------------------------------------------
; If the call worked, then the spec was a directory.
; Open the . (current directory) entry, filling in the
; XDIR structure.
;------------------------------------------------------
        MOV     AH,11H          ;Find First Match
        MOV     DX,OFFSET XFCB  ; for this FCB
        INT     21H             ; Thru DOS

        OR      AL,AL           ;AL=0 if successful
        JZ      M_6
```

Listing 2.1 *continued*

```
;------------------------------------------------------
; The spec was a root directory (since we CD'd
; successfully to it), but not a subdir (since it
; didn't contain a "." file entry). Since we can't walk
; the root directory, command line must be in error.
;------------------------------------------------------
        CALL    RESTORE_DIR             ;Restore dir
        CALL    RESTORE_DRIVE           ; and drive
        MOV     DX,OFFSET NOROOT$       ;Error message
        JMP     M_ERR
;------------------------------------------------------
; The spec wasn't a directory since the CD call failed.
; Assume that it's a file and parse the spec backwards
; to the first backslash, then try to set the resultant
; directory.
;------------------------------------------------------
M_3A:
        MOV     CX,DI           ;From end of cmd line
        SUB     CX,[FILESPEC]   ; subtract start => len
M_3B:
        DEC     DI                ;Back up a char
        CMP     BYTE PTR [DI],"\" ; is it a backslash?
        LOOPNE  M_3B
;------------------------------------------------------
; If we exited the loop because we ran out of chars
; (CX=0) then assume that only a filename was present.
; Otherwise, a backslash was found.
;------------------------------------------------------
        JE      M_4A            ;Jmp if \

        DEC     DI                ;Cancels future INC
        JMP     SHORT M_5A
;------------------------------------------------------
; DI now points to the last backslash in the pathname.
; Overwrite it with a 0 to create an ASCIIZ path and
; an ASCII filename. Handle the special case when the \
; indicates the root dir. Change current dir.
;------------------------------------------------------
M_4A:
        MOV     BYTE PTR [DI],0 ;Change path to ASCIIZ
        MOV     DX,[FILESPEC]   ;Was start of filespec
        CMP     DI,DX           ; that backslash?
        JNE     M_4B

        DEC     DX                  ;Back pointer up
        MOV     BYTE PTR [DI-1],"\" ;Create a root spec
M_4B:
```

Listing 2.1 *continued*

```
;-------------------------------------------------------
; Separate the path from the filespec. If the path
; string is a single backslash, handle it as a special
; case. Otherwise, if we still can't change to it,
; then the argument was invalid.
;-------------------------------------------------------
        MOV     AH,3BH          ;Set dir to DS:DX
        INT     21H             ; Thru DOS
        JNC     M_5A
M_4C:
        CALL    RESTORE_DRIVE
        MOV     DX,OFFSET NOTFOUND$ ;Report the error
        JMP     M_ERR
;-------------------------------------------------------
; Parse the filename into the FCB, detecting illegal
; file names.
;-------------------------------------------------------
M_5A:
        MOV     CX,9             ;8 chars + dot max
        LEA     SI,[DI+1]        ;Point past \
        MOV     DI,OFFSET XFCB+8 ;Name field=dest
M_5B:
        LODSB

        CMP     AL,"."           ;Was it a dot?
        JNE     M_5C

        MOV     CX,3              ;3 ext chars max
        MOV     DI,OFFSET XFCB+16 ;Extension dest
        JMP     M_5B
M_5C:
        OR      AL,AL           ;AL=0 is end of data
        JZ      M_5D

        STOSB
        LOOP    M_5B            ;Continue parsing

        CMP     BYTE PTR [SI],0 ;Shouldn't be any left
        JNE     M_5E
M_5D:
;-------------------------------------------------------
; Open the file using the find first function.
;-------------------------------------------------------
        MOV     AH,11H          ;Find First Match
        MOV     DX,OFFSET XFCB  ; for this FCB
        INT     21H             ; Thru DOS

        OR      AL,AL           ;AL=0 if successful
        JZ      M_5F
```

Listing 2.1 *continued*

```
;-----------------------------------------------------
; Back out and restore if failure.
;-----------------------------------------------------
        CALL    RESTORE_DIR
M_5E:
        CALL    RESTORE_DRIVE
        MOV     DX,OFFSET NOTFOUND$     ;Report error
        JMP     M_ERR
M_5F:
;-----------------------------------------------------
; The starting cluster number is in the XFCB.
; Determine drive parameters.
;
; BPS - Bytes per DOS sector. So we can read the FAT
;       using Int 25h. DOS supports only 512.
;
; SPC - Cluster mask = sectors per cluster - 1. Used to
;       determine actual disk space used by the
;       file/directory.
;
; FFS - First FAT Sector. The logical sector number
;       that contains the first sector of the FAT.
;
; FDS - First Data Sector. The logical sector number of
;       the first data sector on the disk.
;
; MCN - Maximum cluster number. So we can determine if
;       a FAT entry is invalid.
;
; FES - FAT entry size. This is 12 or 16 bits. Needed
;       to calculate the next entry in an allocation
;       chain. Calculated indirectly.
;-----------------------------------------------------
; Use the undocumented function Get Default DPB to
; examine the current drive. Copy the needed info to
; local storage.
;-----------------------------------------------------
M_6:
        MOV     DI,OFFSET DRIVE_DATA    ;ES:DI->dest

        MOV     AH,1FH          ;Get default DPB
        INT     21H             ; Undoc DOS fn
ASSUME  DS:NOTHING              ;Changes DS

        MOV     AX,DS:[BX][2]   ;Bytes/sector
        STOSW                   ;Write to ES:DI

        MOV     AL,DS:[BX][4]   ;Sector mask
        INC     AL              ; +1=SPC
        SUB     AH,AH           ;Zero extend it
        STOSW
```

Listing 2.1 *continued*

```
        MOV     AX,DS:[BX][6]   ;# reserved sectors
        STOSW                   ; = first FAT sector

        MOV     AX,DS:[BX][0BH] ;First Data Sector
        STOSW

        MOV     AX,DS:[BX][0DH] ;Max cluster number
        STOSW

        PUSH    CS              ;Put CS segment
        POP     DS              ; into DS
ASSUME  DS:CSEG
;------------------------------------------------------
; IF (# clusters<4087) THEN FAT=12-bit ELSE FAT=16-bit.
;------------------------------------------------------
        CMP     AX,4086         ;Check # of clusters
        JA      M_7
        MOV     BYTE PTR [FES],FES_12    ;FAT type flag
M_7:
;------------------------------------------------------
; If drive has more than 65535 total sectors, must use
; huge protocols. Multiply number of clusters (in AX)
; by sectors per cluster. DX will hold the high-order
; word of the 32-bit result. If the disk is huge, DX
; will be non-zero.
;------------------------------------------------------
        MOV     DX,[SPC]        ;Sectors per cluster
        MUL     DX              ;* # clusters
        MOV     [HUGEDISK],DX   ;Use DX as flag
;------------------------------------------------------
; A 12-bit FAT entry can span a logical sector
; boundary. Make sure we've got enough memory to hold
; two FAT sectors, used while walking the FAT chain.
;------------------------------------------------------
        MOV     AX,SP           ;Get the end of segment
        SUB     AX,OFFSET FATBUF+512 ;sub prog/stack

        SHR     AX,1            ;Divide by 2 to see if
        CMP     AX,[BPS]        ; enough for 2 sectors?
        JAE     M_8

        MOV     DX,OFFSET MEMERR$ ;Memory error message

        MOV     AH,9            ;Display string
        INT     21H             ; thru DOS

        CALL    RESTORE_DIR     ;Restore old dir
        CALL    RESTORE_DRIVE   ; and drive
        JMP     M_EXIT
M_8:
```

Listing 2.1 *continued*

```
;----------------------------------------------------
; Now walk the spec's allocation chain through the FAT,
; printing out a status report as we go.
;----------------------------------------------------
        MOV     AH,9            ;Display string
        MOV     DX,OFFSET HEAD$ ; column heads
        INT     21H             ; Thru DOS
;----------------------------------------------------
; Point to the first cluster in the first fragment.
;----------------------------------------------------
        MOV     AX,[FCLUS]      ;Start here
;----------------------------------------------------
; Verify that the first cluster number in AX is valid.
;----------------------------------------------------
        CALL    VERIFY_CLUSTER  ;Checks AX
        JNC     M_9B
M_9A:
        CALL    RESTORE_DIR     ;Restore old dir
        CALL    RESTORE_DRIVE   ; and drive
        JMP     M_ERR
;----------------------------------------------------
; Display the current cluster and first sector.
;----------------------------------------------------
M_9B:
        PUSH    AX
        PUSH    DX
        SUB     SI,SI           ;Show starting sector
        CALL    SHOW_CLUSTER    ;Display cluster info
        JC      M_9A
;----------------------------------------------------
; Save current cluster. Get next cluster in the chain.
;----------------------------------------------------
M_9C:
        MOV     DI,AX           ;Prev=current
        CALL    NEXT_CLUSTER    ;Get next in chain
        JC      M_9A
;----------------------------------------------------
; Is the returned cluster valid?
;----------------------------------------------------
        CALL    VERIFY_CLUSTER
        JC      M_9A
;----------------------------------------------------
; Is the current cluster the end-of-chain?
;----------------------------------------------------
        MOV     BL,0F8H         ;(F)FF8 or greater
        MOV     BH,[FES]        ; is end-of-chain
        CMP     AX,BX
        JB      M_9D
```

Listing 2.1 *continued*

```
;------------------------------------------------------
; Print prev cluster, ending sector.
;------------------------------------------------------
        MOV     AX,DI           ;Previous cluster
        MOV     SI,1            ;Ending sector
        CALL    SHOW_CLUSTER
        JC      M_9A
        JMP     SHORT M_10      ;Print summary
;------------------------------------------------------
; Increment the cluster count.
;------------------------------------------------------
M_9D:
        INC     WORD PTR [NCLUS]  ;Count this cluster
;------------------------------------------------------
; Is the new cluster contiguous with the previous one?
;------------------------------------------------------
        MOV     SI,DI           ;Previous
        INC     SI              ; +1 gives
        CMP     AX,SI           ; contiguous clusters?
        JE      M_9C
;------------------------------------------------------
; Print previous cluster, ending sector.
;------------------------------------------------------
        XCHG    AX,DI           ;Get prev cluster
        MOV     SI,1            ;Show ending sector
        CALL    SHOW_CLUSTER
        JC      M_9A

        XCHG    AX,DI           ;Restore order
;------------------------------------------------------
; Increment the fragment count.
;------------------------------------------------------
        INC     WORD PTR [NFRAG]
        JMP     M_9B
;------------------------------------------------------
; Summarize the spec's cluster info.
;------------------------------------------------------
M_10:
        MOV     AH,9                    ;Display string
        MOV     DX,OFFSET NUMCLUS$
        INT     21H                     ; thru DOS

        MOV     AX,WORD PTR [NCLUS]     ;Print # clus
        CALL    HEXWORD                 ; 4-digit hex

        MOV     AH,9                    ;Display string
        MOV     DX,OFFSET NUMFRAGS$
        INT     21H                     ; thru DOS

        MOV     AX,WORD PTR [NFRAG]     ;Print # frags
        CALL    HEXWORD                 ; 4-digit hex
```

Listing 2.1 *continued*

```
        MOV     AH,9                    ;Display string
        MOV     DX,OFFSET PERCENT$
        INT     21H                     ; thru DOS
;------------------------------------------------------
; Calculate %fragmentation = 100*(NF-1)/(NC-1).
;------------------------------------------------------
        SUB     AX,AX           ;0% fragmented

        MOV     BX,[NCLUS]      ;Div by NCLUS
        DEC     BX              ; minus 1
        JZ      M_11B           ;Only 1 clus?

        MOV     AX,[NFRAG]      ;# fragments
        DEC     AX              ; minus 1

        CMP     AX,BX           ;Completely fragmented?
        JNE     M_11A

        MOV     AH,2            ;Print 1st digit of 100
        MOV     DL,"1"
        INT     21H             ; thru DOS

        SUB     AL,AL           ;"00" digits
        JMP     SHORT M_11B
M_11A:
        MOV     DX,100          ;Result
        MUL     DX              ; in DX:AX
        DIV     BX              ; in AX
M_11B:
        AAM                     ;Split digits
        OR      AX,3030H        ;Make ASCII
        MOV     DL,AH           ;DL = 10s digit
        PUSH    AX              ;Save 1s
        MOV     AH,2            ;Display char
        INT     21H             ; thru DOS

        POP     DX              ;DL = 1s digit
        MOV     AH,2            ;Display char
        INT     21H             ;thru DOS
;------------------------------------------------------
; Restore original directory and drive.
;------------------------------------------------------
        CALL    RESTORE_DIR
        CALL    RESTORE_DRIVE
        JMP     M_EXIT

MAIN    ENDP
```

Listing 2.1 *continued*

```
;======================================================
; CHANGE_DRIVE
; Isolate a drive spec (if present) and make it the new
; default drive.
;------------------------------------------------------
; Entry: none
; Exit:
;       NC: success
;       CY: DS:DX -> error message
;------------------------------------------------------
CHANGE_DRIVE    PROC    NEAR
        ASSUME  CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG
;------------------------------------------------------
; DOS automatically determines the validity of the
; first command line argument's drive spec and puts the
; result in AL. AL=FFh if the drive was invalid.
;------------------------------------------------------
        CMP     AL,0FFH         ;Check for invalid drv
        JNE     CD_1

        MOV     DX,OFFSET INVDRV$       ;Error message
        STC                             ;Carry flag on
CD_EXIT:
        RET
;------------------------------------------------------
; Fetch and save the current drive for use during disk
; reads or if we have to change the default drive and
; later restore it.
;------------------------------------------------------
CD_1:
        MOV     AH,19H          ;Get current drive #
        INT     21H             ; Thru DOS

        MOV     [OLDDRV],AL     ;Save current drive
        MOV     [TGTDRV],AL     ;Initialize new drive
;------------------------------------------------------
; If a drive was specified on the command line, DOS
; will have already parsed it and placed a drive number
; in the first default FCB: 0=none, 1=A, etc.
;------------------------------------------------------
        MOV     DL,DS:[5CH]     ;Get drive from FCB
        DEC     DL              ;If 0,turns sign bit on
        JS      CD_2

        MOV     [TGTDRV],DL     ;Save working drive

        MOV     AH,0EH          ;Select current drive
        INT     21H             ; Thru DOS
CD_2:
        CLC
        JMP     CD_EXIT
```

Listing 2.1 *continued*

```
CHANGE_DRIVE    ENDP

;=======================================================
; PARSE_CMDLINE
; Isolate the command line argument with pointers.
;-------------------------------------------------------
; Entry: none
; Exit:
;       NC:
;         DS:SI -> start of string
;         DS:DI -> end of string
;       CY:
;         DS:DX -> error message
;-------------------------------------------------------
PARSE_CMDLINE   PROC    NEAR
        ASSUME  CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG
;-------------------------------------------------------
; A command line arg must be specified.
;-------------------------------------------------------
        CLD                     ;String moves forward

        MOV     SI,81H          ;PSP:81 = command tail

        SUB     CX,CX           ;Clear CX
        ADD     CL,[SI-1]       ;# chars in cmd tail
        JNZ     PC_2A
PC_1:
        MOV     DX,OFFSET USAGE$ ;Error message
PC_ERR:
        STC                     ;Set carry flag
PC_EXIT:
        RET
;-------------------------------------------------------
; Scan off leading blanks.
;-------------------------------------------------------
PC_2A:
        LODSB                   ;Get a character
        CMP     AL,BLANK        ;Skip blanks
        JNE     PC_2B
        LOOPE   PC_2A           ;Continue scan
        JCXZ    PC_1            ;Jump if all blanks
PC_2B:
```

Listing 2.1 *continued*

```
;-------------------------------------------------------
; Save first non-blank. Find the end of the argument.
;-------------------------------------------------------
        DEC     SI              ;Point SI to first char
        MOV     DI,SI           ; and save in DI
PC_2C:
        LODSB                   ;Get char
        CMP     AL,BLANK        ;Skip non-blanks
        JE      PC_2D
        LOOP    PC_2C

        INC     SI              ;Cancel next DEC
PC_2D:
        DEC     SI              ;Back up to blank
        MOV     BYTE PTR [SI],0 ;Make ASCIIZ
        XCHG    SI,DI           ;SI=start DI=end (used)
;-------------------------------------------------------
; If a drivespec appears, remove it.
;-------------------------------------------------------
        CMP     BYTE PTR [SI+1],":"     ;Colon?
        JNE     PC_3

        ADD     SI,2            ;Skip "d:" in spec

        CMP     BYTE PTR [SI],0 ;Spec empty now?
        JE      PC_1
PC_3:
        CLC
        JMP     PC_EXIT

PARSE_CMDLINE   ENDP

;=======================================================
; RESTORE_DRIVE
; Change back to the original default drive.
;-------------------------------------------------------
; Entry: none
; Exit: CF returned by DOS
;-------------------------------------------------------
RESTORE_DRIVE   PROC    NEAR
        ASSUME  CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG

        MOV     DL,[OLDDRV]     ;Old drive
        CMP     DL,[TGTDRV]     ; same as current?
        JE      RD_EXIT

        MOV     AH,0EH          ;Set current drive
        INT     21H             ; Thru DOS
RD_EXIT:
        RET

RESTORE_DRIVE   ENDP
```

Listing 2.1 *continued*

```
;======================================================
; RESTORE_DIR
; Restore the original default directory.
;------------------------------------------------------
; Entry: none
; Exit: CF returned by DOS
;------------------------------------------------------
RESTORE_DIR     PROC    NEAR
        ASSUME  CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG

        MOV     AH,3BH            ;Set directory
        MOV     DX,OFFSET OLDDIR  ; to what we found
        INT     21H               ; Thru DOS
        RET

RESTORE_DIR     ENDP

;======================================================
; VERIFY_CLUSTER
; Verify that the cluster number in AX is valid for the
; current drive.
;------------------------------------------------------
; Entry:
;       AX = current cluster number
; Exit:
;       NC: success
;       CY: DX = offset of error message
;------------------------------------------------------
; Changes: BX DX
;------------------------------------------------------
VERIFY_CLUSTER  PROC    NEAR
        ASSUME  CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG
;------------------------------------------------------
; Error if cluster number is 0 or 1.
;------------------------------------------------------
        MOV     DX,OFFSET INVCLUS$      ;Error message
        CMP     AX,2                    ;0 or 1?
        JB      VC_EXIT                 ;JB=JC
;------------------------------------------------------
; AX == (F)FF7 indicates a bad cluster. Return error.
; AX >= (F)FF8 terminates the allocation chain. Return
;              valid.
;------------------------------------------------------
VC_1:
        MOV     DX,OFFSET BADCLUS$      ;Error message

        MOV     BL,0F7H                 ;Bad cluster
        MOV     BH,[FES]                ;Upper mask

        CMP     AX,BX                   ;Check value
        JA      VC_EXIT                 ;CF=0
```

Listing 2.1 *continued*

```
        JB      VC_2
        STC                             ;CF=1
VC_EXIT:
        RET
;-------------------------------------------------------
; Check for invalid cluster number.
;-------------------------------------------------------
VC_2:
        CMP     AX,[MCN]        ;Max cluster number
        CMC                     ;Reverse CF
        JMP     VC_EXIT

VERIFY_CLUSTER  ENDP

;=======================================================
; SHOW_CLUSTER
; Display cluster number and translate first or last
; logical sector in the cluster.
;-------------------------------------------------------
; Entry:
;       AX = current cluster number (assumed valid)
;       SI = 0, display 1st sector in cluster
;            otherwise, display last sector in cluster
; Exit:
;       NC: success
;       CY: DX = offset of error message
;-------------------------------------------------------
; Changes: BX DX DI BP
;-------------------------------------------------------
SHOW_CLUSTER    PROC    NEAR
        ASSUME  CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG

        PUSH    AX              ;Save register
;-------------------------------------------------------
; A value of (F)FF8h or greater terminates the chain.
;-------------------------------------------------------
        SUB     DX,DX           ;Assume end of chain

        MOV     BX,0FFF7H       ;16-bit value
        AND     BH,[FES]        ;Mask if needed

        CMP     AX,BX           ;Check value
        JB      SC_1
SC_ERR:
        STC                     ;CF=1 is error
SC_EXIT:
        POP     AX              ;Restore register
        RET
```

Listing 2.1 *continued*

```
;-------------------------------------------------------
; Show the cluster number as 4-digit hex.
;-------------------------------------------------------
SC_1:
        CALL    HEXWORD                 ;Display AX

        PUSH    AX                      ;Save cluster

        MOV     AH,9                    ;Display string
        MOV     DX,OFFSET SPACE$        ; 1 space
        INT     21H                     ; thru DOS

        POP     AX                      ;Restor cluster
;-------------------------------------------------------
; Display the first sector in the cluster (SI=0) or the
; last sector in the cluster (SI != 0) as 32-bit hex.
;-------------------------------------------------------
        SUB     AX,2            ;Disregard ID bytes

        MOV     BX,[SPC]        ;Sectors per cluster
        MUL     BX              ;DX:AX=sectors

        ADD     AX,[FDS]        ;Offset from FDS
        ADC     DX,0            ;In case 32-bit #

        OR      SI,SI           ;SI = 0 for 1st sector
        JZ      SC_2

        DEC     BX
        ADD     AX,BX           ;Add cluster length
        ADC     DX,0            ;In case 32-bit #
SC_2:
        XCHG    AX,DX           ;Get high-order word
        CALL    HEXWORD         ; display it
        MOV     AX,DX           ;Then low-order word
        CALL    HEXWORD

        MOV     DX,OFFSET SPACE$        ; 1 space
        OR      SI,SI
        JZ      SC_3

        MOV     DX,OFFSET CRLF$
SC_3:
        MOV     AH,9            ;Display string
        INT     21H             ; thru DOS

        CLC                     ;Say success
        JMP     SC_EXIT

SHOW_CLUSTER    ENDP
```

Listing 2.1 *continued*

```
;=======================================================
; NEXT_CLUSTER
; Given a cluster number, reads the appropriate FAT
; sector into memory, looks up the entry in the FAT,
; and retrieves the number of the next cluster in the
; chain. Assumes the passed cluster number is valid.
;-------------------------------------------------------
; Entry:
;       AX = valid cluster number
; Exit :
;       NC: AX = next cluster number
;       CY: DX = error message
;-------------------------------------------------------
; Changes: AX BX CX DX BP
;-------------------------------------------------------
NEXT_CLUSTER    PROC    NEAR
        ASSUME  CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG
;-------------------------------------------------------
; From the cluster number, determine in which sector of
; the FAT this cluster is located. 12-bit and 16-bit
; FATs are decoded differently.
;-------------------------------------------------------
        SUB     DX,DX           ;Prepare for long DIV

        CMP     [FES],FES_12    ;Check FAT type
        JNE     NC_1A
;-------------------------------------------------------
; Decode a 12-bit FAT entry.
; 1. Translate the 12-bit cluster number into byte
;    offset into FAT. Since the maximum cluster number
;    for a 12-bit FAT is 4086, and since each FAT entry
;    is 1.5 bytes, the maximum offset into the FAT is
;    6129 [17F1h] bytes. This always fits a word.
;-------------------------------------------------------
        SUB     BP,BP           ;Create a zero

        MOV     BX,AX           ;Multiply
        ADD     AX,AX           ; AX
        ADD     AX,BX           ; by 3
        SHR     AX,1            ;Divide by 2

        RCL     BP,1            ;Save carry flag state
        JMP     SHORT NC_1B
```

Listing 2.1 *continued*

```
;-------------------------------------------------------
; Decode a 16-bit FAT entry.
; 1. Translate cluster number into a byte offset into
;    FAT. Note that while the cluster number can never
;    exceed 16 bits, the byte offset into the FAT can.
;-------------------------------------------------------
NC_1A:
        SHL     AX,1            ;Multiply AX by 2
        RCL     DX,1            ;Move CY into DX
NC_1B:
;-------------------------------------------------------
; 2. Divide the offset by the number of bytes in a
;    sector to determine into which sector of the FAT
;    this offset points.
;-------------------------------------------------------
        MOV     CX,[BPS]        ;Bytes per sector
        DIV     CX              ;AX=sects, DX=bytes
;-------------------------------------------------------
; 3. If the required FAT sector (in AX) isn't the first
;    one in memory, load it and the one following it.
;    (This accommodates a 12-bit FAT entry that spans a
;    sector boundary.)
;-------------------------------------------------------
        CMP     AX,[FATSECT]    ;Is this sect in mem?
        JE      NC_2A
;-------------------------------------------------------
; Read in the FAT sectors starting at AX.
;-------------------------------------------------------
        MOV     [FATSECT],AX    ;Say this one's here

        CALL    READ_FAT
        JNC     NC_2A

        MOV     DX,OFFSET FATERR$ ;Report err, CY set
NC_EXIT:
        RET
;-------------------------------------------------------
; 4. Read a word at the specified offset (in DX). If
;    16-bit FAT, we're done. For a 12-bit FAT, mask off
;    the appropriate bits.
;-------------------------------------------------------
NC_2A:
        MOV     BX,DX                     ;Offset in DX
        MOV     AX,WORD PTR [FATBUF][BX]  ;Get the word

        CMP     [FES],FES_12            ;Check FAT type
        JNE     NC_2C
```

Listing 2.1 *continued*

```
;-------------------------------------------------------
; For 12-bit FATs, if the original cluster was even,
; use the lower 12 bits of the word. If the original
; cluster was odd, use the upper 12 bits of the word.
;-------------------------------------------------------
        OR      BP,BP           ;Was mult whole word?
        JZ      NC_2B

        MOV     CL,4            ;Shift count
        SHR     AX,CL           ;Use upper 12 bits
        JMP     SHORT NC_2C
NC_2B:
        AND     AH,0FH          ;Use lower 12 bits
NC_2C:
        CLC                     ;Say success
        JMP     NC_EXIT

NEXT_CLUSTER    ENDP

;=======================================================
; READ_FAT
; This routine reads a sector from the drive into the
; FATBUF. It adjusts automatically for huge (type 6)
; drives.
;-------------------------------------------------------
; Entry:
;       AX = Number of the FAT sector to read.
; Exit: None
;-------------------------------------------------------
; Changes: AX BX CX
;-------------------------------------------------------
READ_FAT        PROC    NEAR
        ASSUME  CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG

        PUSH    DX              ;Save register
;-------------------------------------------------------
; Convert the relative FAT sector to an absolute
; logical sector number.
;-------------------------------------------------------
        ADD     AX,[FFS]        ;Convert to disk sector
;-------------------------------------------------------
; If not a huge disk, use the normal function protocol.
;-------------------------------------------------------
        CMP     [HUGEDISK],0
        JNE     RF_1A

        MOV     DX,AX             ;Starting sector
        MOV     AL,[TGTDRV]       ;Read from this drive
        MOV     BX,OFFSET FATBUF  ;Put data here
        MOV     CX,2              ;Read two sectors
        JMP     SHORT RF_1B
```

Listing 2.1 *continued*

```
;-------------------------------------------------------
; Use the type 6 disk protocol. Because it's a 16-bit
; FAT, we only have to read one sector.
;-------------------------------------------------------
RF_1A:
        MOV     [FIRST_SECTOR],AX
        MOV     [BUF_SEG],DS
        MOV     AL,[TGTDRV]         ;Read from this drive
        MOV     BX,OFFSET DISK_PACKET
        MOV     CX,-1               ;Signal type 6
RF_1B:
        INT     25H               ;Direct disk read
        POP     DX                ;Discard old flags
        POP     DX                ;Restore register
        RET

READ_FAT        ENDP

;=======================================================
; HEXWORD - Write AX as 4 hex digits to std out
;-------------------------------------------------------
; Entry:
;       AX = value to display
; Exit: None
;-------------------------------------------------------
; CHANGES: None
;-------------------------------------------------------
HEXWORD         PROC    NEAR
        ASSUME  CS:CSEG, DS:CSEG, ES:NOTHING, SS:CSEG

        PUSH    CX                ;Save registers
        PUSH    DX

        MOV     CX,4              ;Number digits
H_1:
        PUSH    CX                ;Save digit count
        MOV     CL,4              ;Rotate digit
        ROL     AX,CL             ; into low 4 bits
        POP     CX                ;Restore digit count
```

Listing 2.1 *continued*

```
        PUSH    AX              ;Save argument

        AND     AL,0FH          ;Convert AL to ASCII
        ADD     AL,90H
        DAA
        ADC     AL,40H
        DAA

        MOV     AH,2            ;Display char
        MOV     DL,AL           ; in DL
        INT     21H             ; Thru DOS

        POP     AX              ;Restore argument

        LOOP    H_1

        POP     DX              ;Restore registers
        POP     CX
        RET

HEXWORD         ENDP

;=====================================================================
; Allocate buffers here dynamically.
;---------------------------------------------------------------------
FATBUF  EQU     $

CSEG    ENDS
        END     ENTPT

; End of File
```

Chapter 3

Modifying DOS Boots

Steven K. Graham

DOS has a certain inevitability to it — you turn on the machine and there is the familiar prompt (granted, you may have to wait a few seconds). It is ubiquitous and you may associate the prompt and the MS-DOS operating system with the hardware. This is erroneous. Ultimately, the machine is one of Intel's microprocessors: 8088, 8086, 80286, 80386, or 80486. DOS is *not* integral to the hardware. DOS is simply a program — running other programs loaded from disk and controlling system peripherals by driving I/O ports. DOS is only one of many such programs that *can* be run by the CPU. There are alternatives.

The DOS boot process proceeds through various testing and initialization tasks, eventually reaching the ROM bootstrap loader, which searches system disks for a boot sector. If the bootstrap program fails to find a boot sector, it prints the cryptic message:

```
Non-system disk or disk error
Replace and strike any key when ready
```

If the bootstrap loader finds a boot sector, it copies the sector into memory and transfers control to the disk boot program. If the disk contains a DOS boot sector, the system loads and executes DOS. But the boot sector could contain a program that loads and executes an alternative operating system; or a program that performs various security checks; or (on a disk containing an application or data) a program that provides a more intelligible explanation of why the system hasn't booted and what the user should do (preventing terror from the "disk error" message and explaining that the only problem is an inappropriate floppy in the drive). An extensive diagnostics package would be especially useful as a program initiated from a boot sector. Such a diagnostic program could run and test a malfunctioning system that can still execute ROM BIOS commands, but can't successfully run DOS.

This chapter contains examples of such programs, along with discussion and explanation of background and related material. Three examples will show a range of possibilities for modified boots. The first is a simple boot program used to display a custom message and nothing more. The second example illustrates changes to an existing boot sector that add a layer of system security. The third example demonstrates the techniques for loading and executing a more complex program — the steps needed for a diagnostic program.

Warning!

In case of accidental damage, have a bootable floppy and a disk utility that can repair the boot sector.

If you are unfamiliar with the boot process, read the "Overview of the DOS Boot Process" (see page 50). If you understand the general process, but need a low level description of boot sectors and boot programs, see the section "Boot Sectors, Master Boot Sectors, and Boot Programs" (see page 54).

First Example: Boot Program

The following program can replace a boot sector and allows you to insert a custom message. The program does not include a bootstrap loader, so it will not load system files and start DOS. The program is easily modified to provide an informative message rather than a terse complaint.

Listing 3.1, the program `BOOT360.ASM`, shows the structure of a boot program, particularly the BIOS Parameter Block (BPB), in this case for a 360Kb 5.25 inch floppy. The program ends by requesting a keystroke via `BIOS INT 16h` and then restarting the system, doing a warm boot.

`BOOT360.ASM` and the other programs in this article were written using Borland's Turbo Assembler v1.01. The programs were tested on systems running DOS v3.3: an XT compatible with a Phoenix BIOS and an AT compatible with an AMI BIOS were the target systems.

Alternate BIOS parameter blocks can modify `BOOT360.ASM` for use with other sizes of floppies (see the excerpts in Listing 3.2).

Storing Custom Boot Record to a Disk

Once you've created a custom boot program, you've got to install the program into the boot sectors of the disks you intend to use. The DOS tool `DEBUG` is the simplest way to copy a new boot program onto a disk. The following sequence of commands illustrates the process for a boot program stored as a binary file `BOOT.BIN`. Such a file can be created from an `.EXE` file using the program `EXE2BIN`.

```
C:\DOS> debug
- n boot.bin ; specify file name
- l 0   ; load file at address 0
- w 0 0 0 1 ; from address 0, write to drive 0, sector 0 one sector
- q
C:\DOS>
```

Writing sectors with `DEBUG` is hazardous to the contents of your disks. If you aren't familiar with `DEBUG`, study a thorough discussion (such as in *PC Magazine's DOS Power Tools* by Paul Somerson). The second parameter of `w 0 0 0 1` specifies the destination drive. You can copy the boot program to your B: drive by substituting `w 0 1 0 1` or your C: drive with `w 0 2 0 1`.

Hard Disks and Boot Programs

Hard disks generally don't need any special boot program. The boot record could be replaced by a program similar to the previous example, preventing anyone from booting the system without a bootable floppy. Listing 3.3 shows an altered version of the boot program for a Seagate 4038 hard disk.

Second Example: Changes to Existing Boot Sector — System Security

An alternative change to the boot process introduces a simple pass-key security: on reboot, the appropriate key must be pressed or the boot is aborted. Many boot sectors have some space unused and the additions require only 16 bytes, so the small program can be incorporated into existing boot sectors. Listing 3.4 shows a modified boot program including the pass-key check. To make these changes to your boot record, follow these steps.

1. Determine whether there is sufficient room: Load the boot record with debug and look at the end of the sector, immediately prior to the signature. Often there will be enough bytes to insert the pass-key program. In `DEBUG`, the commands:

   ```
   - l 0 2 0 1
   - d 180
   ```

   will load the boot record and display the final 80 (hex) bytes.
2. The first instruction in the first byte of the boot record must be changed to jump to the pass-key code. `DEBUG` commands are:

   ```
   -a 0
   xxxx:0000 jmp 1EE <cr>
   xxxx:0003 <cr>
   -
   ```

   Remember the destination of the original `jmp` instruction (often `jmp 36`). The pass-key code will jump to the same address after the right key is pressed.

3. You must locate an error routine to use if the wrong key is pressed. Display the sector and find the starting address of an error message. Use the `u` command to unassemble code in the sector, searching for:

   ```
   mov si,7xxx
   ```

   where `7C00+offset` of the error message `= 7xxx`, followed by a `jmp` or call to a portion of code such as:

   ```
   lodsb
   or al,al
   jz nnnn ; ret instruction or continuation of error handler
   mov bx,7
   mov ah,0E
   int 10
   jmp kkkk ; jmp back to lodsb instruction
   ```

   This code writes an error message to the display. The pass-key code will branch to an existing error message if an incorrect key is pressed. Remember the address of the `mov si,7xxx` instruction, for use as your error routine.
4. Determine which key to use as the pass-key that permits the system to boot. Find the scan code returned in register `ax` for that key. ESC returns `011Bh`.
5. Enter the following debugger commands, changing the addresses and scan code as appropriate.

   ```
   - a 1EE
   xxxx:01EE xor ax,ax
   xxxx:01F0 int 16
   xxxx:01F2 cmp ax,011B ; substitute scan code here
   xxxx:01F5 jz 1FA ; short conditional jump if OK
   xxxx:01F7 jmp 0C2 ; substitute error routine address
   xxxx:01FA jmp 036 ; substitute initial jump destination
   xxxx:01FD <cr>
   -
   ```

6. The addresses must be adjusted to the particular disk and boot record used. After the above changes have been assembled into the boot record, write the record back to the disk: `- w 0 2 0 1`. Listing 3.4 shows changes to the example boot program, but the changes should be made to your existing boot record.

Third Example: More Complex Programs

The next two programs make more elaborate use of the boot process. `BOOTRUN` is a boot program that loads a second program, `BOOTSYS`, much as the normal bootstrap loader brings in `IO.SYS`. Listings 3.5 and 3.6 illustrate such a process. A custom version can be created by writing your own `BOOTSYS.BIN` program. `EXE2BIN` can be used to convert the programs from `.EXE` to `.BIN`.

Caveats

In the example above, `BOOTRUN` attempts to read the `BOOTSYS` sectors only once. For robustness, bootstrap loaders normally make several attempts (five is a good number). A simple loop wrapped around the code that loads `BOOTSYS` would make `BOOTRUN` more robust.

`BOOTRUN`, as written, requires that `BOOTSYS` be the first file on the disk and that `BOOTSYS` be stored in contiguous sectors. `BOOTRUN` also imposes a size restriction on `BOOTSYS`. Changes to `BOOTRUN`, could make it more flexible. By reading the FATs (see Chapter 2) and root directory, `BOOTRUN` can be modified to load and execute a program of arbitrary size from anywhere on the disk.

The PC boot process is complex, but it can be modified to serve the needs of developers. These programs illustrate the fundamentals of using the bootstrap loader and ROM BIOS to provide varied services before DOS starts running.

Overview of the DOS Boot Process

When a PC is turned on, the CPU begins executing instructions from a location fixed when the chip was designed. The bulk of memory available to the processor will be volatile (e.g., dynamic RAM) — the processor doesn't know, can't tell, and must not rely on RAM contents — *when it is starting up*. Since the CPU always begins execution at a fixed location, non-volatile memory (ROM) is mapped to that location.

Whenever a member of Intel's 80x86 family is first turned on, the code segment register(`CS`) is set to `0F000h` and the instruction pointer (`IP`) to `0FFF0h`. This fixed location (`0F000:FFF0h`, `0FFFF0h absolute`) is mapped to ROM BIOS and a system restart routine. `0FFFF0h` is only 16 bytes away from the upper memory limit for the 8088 and 8086, with their 20-bit address space. The system restart routine immediately jumps to initialization code, the *Power-on Self Test* (*POST*) routines, and the ROM bootstrap loader. In later machines (80386), initial *CS:IP* contents are *000F:FFF0h*, but the first instruction fetched is from absolute address *0FFFFFFF0h*, just 16 bytes shy of the upper address limit for the 80386. Until the *CS* register is first changed, instruction access is from segment *0FFFFh*. The 80386 begins operation in real address mode, so after the first change to the *CS* register, the CPU is restricted to the lower 1Mb of the address space.

Warm, Cold and Other Boots

A system boot may be either "warm" or "cold." When you turn on your system, you initiate a cold boot. Typically, using the "reset" button on your computer also causes a cold boot. Warm boots are usually started by typing Ctrl-Alt-Del. Warm boots perform less thorough system testing than cold boots before proceeding to load the operating system. These reboots are initiated by branching to *0F000:FFF0h*, the power-on location. Since both warm and cold boots execute the same code, the value at memory location *0040:0072h* flags the type of boot to perform. If *1234h* has been stored at *0040:0072h* then a warm boot is performed and some hardware and memory tests are omitted. Some systems can store *4321h* into *0040:0072h* and preserve memory contents (except as required to load the operating system) while restarting. Another alternative is to directly invoke *INT 19h*, the ROM BIOS bootstrap loader. *INT 19h* bypasses the *POST* and system initialization completely, but loads fresh copies of the system files into memory and starts DOS.

POST

POST, an acronym for *Power On Self Test*, is a sequence of initialization and testing routines executed at boot time. These are the routines that give you messages such as "floppy disk controller not working." Since the introduction of the AT, *POST* has been a part of the boot sequence. *POST* is responsible for the "beep" codes that indicate errors when booting. Systems checked often include: motherboard, memory, keyboard, display and disk adapters, and coprocessors.

Initialization

Part of the boot process includes initialization of various data areas in low memory. The interrupt vector table (*0:0000h - 0:03FFFh*) and ROM

BIOS tables (*0:0400h - 0:04FFh*) are initialized. The boot process also searches for any ROM extensions between *0A0000h* and *0F0000h*, triggering initialization routines in the extensions.

The ROM Bootstrap Loader

The ROM bootstrap loader is the heart of the boot sequence. After peripherals identified in ROM BIOS have been identified, tested and initialized, the system must load the boot sector and system files from disk. These programs are too large or too variable (depending on system configuration) or too infrequently used to be included in ROM. The bootstrap loader loads the disk boot program and systems files from disk. Since the operating system is not yet available, loading of the boot programs must be done at a very basic level — only what is known about the disk in ROM BIOS can be used. ROM BIOS provides interrupt routines that can read specific sectors from disk into memory (BIOS *INT 13h*, function *2*), so the boot program resides in a known location, the first sector of a bootable disk, referred to as the boot sector. The ROM bootstrap loader tries to read a boot sector first from floppy drive A:, and, failing that, tries to read from hard drive C:, if the system has a hard drive installed.

Boot Program

The boot sector contains information about the system and disk organization (the BPB), as well as a short program (the boot program or disk bootstrap loader). The boot program uses BIOS services available in ROM and information from the BPB to load the *IO.SYS* file, which contains the additional BIOS routines and the *SYSINIT* module that controls the rest of system restart.

IO.SYS

IO.SYS (or *IBMBIO.COM* in PC-DOS) contains various BIOS routines not supplied in ROM, usually unique to a particular manufacturer or system configuration or initialization code, required only at system restart. A second module in *IO.SYS* is *SYSINIT* which relocates *MSDOS.SYS*, the DOS kernel, to its final location. *IO.SYS* is loaded by the disk bootstrap loader contained in the boot sector of the boot disk. *IO.SYS* must be the first entry in the root directory and must be stored in contiguous clusters at the beginning of the disk's data area. *MSDOS.SYS* is loaded either by the

DISK bootstrap loader or by `IO.SYS`. After `MSDOS.SYS` is positioned, `SYSINIT` executes an initialization routine in `MSDOS.SYS`. Subsequently, `SYSINIT` can use DOS services to find and load `CONFIG.SYS`. Commands in `CONFIG.SYS` are executed to allocate buffers and load and initialize device drivers. `SYSINIT` uses the `DOS EXEC` function to load and jump to the command interpreter (or shell), usually `COMMAND.COM`. `COMMAND.COM` processes the `AUTOEXEC.BAT` file, finally presenting the `C:>` when system initialization and loading is complete.

MSDOS.SYS

`MSDOS.SYS` (`IBMDOS.COM` in PC-DOS) comprises the DOS kernel — routines providing the essential DOS services that remain memory resident. An initialization routine is invoked by `SYSINIT` which creates DOS tables, buffers and sets up the DOS interrupt vectors (`20h - 2Fh`). MS-DOS then returns control to `SYSINIT`, which handles `CONFIG.SYS` and `EXEC`'s `COMMAND.COM`.

COMMAND.COM

`COMMAND.COM`, while the prominent aspect of "MS-DOS," is not intrinsic to DOS. `COMMAND.COM` is merely a shell or command interpreter which can be readily replaced. `COMMAND.COM` provides much of the user interface that is familiar as DOS, but the underlying service routines form the "real" DOS, and the features and limitations of these routines shape the character of all DOS shells and applications.

Boot Failure ...

Systems fail to boot for a variety of reasons. Absence of a bootable system disk is the most common. Similarly, corruption of the boot sector or problems with reading from the boot disk can also abort the boot. Usually, if a system fails to boot, a non-system floppy has been left in the A: drive of a system that normally boots from the C: drive. If some failure causes

```
Boot Sector
FAT1
FAT2
Root Directory
```

Figure 3.1 ***Logical disk contents.***

a C: drive not to boot, often the machine can still be booted using a system disk in floppy drive A:. Then, one of various disk utility programs can be used in an attempt to make the C: drive bootable once more.

Overview of Disk Organization

Floppy disks have a logical structure that corresponds to their physical structure, but since a single physical hard disk may be used as more than one logical disk, hard disk structure is more complicated. Every logical disk (all floppies, hard disk partitions) consist of boot sector, FATs, root directory, and data. Note that "data" in this context refers not only to actual data files, but to all files and subdirectories, including applications and other executable programs not often thought of as data.

Hard disks include an initial master boot sector, containing the partition table, followed by zero or more reserved sectors, and then followed by one to four logical disks.

Boot Sectors, Master Boot Sectors, and Boot Programs

The ROM bootstrap loader checks first the A: drive and then the C: drive (if installed) for a bootable disk and a boot sector. The boot sector consists of the boot record, containing a BPB that describes the boot drive, and a boot program consisting of a bootstrap loader that continues the boot process by loading in various system files. The boot program is followed by a trailing signature of two bytes, `55h`, `0AAh` (the final two bytes of the sector).

```
Master Boot Sector
    - contains Partition Table
Reserved Sectors
Logical Disk
    - contains:
    Boot Sector
    FAT1
    FAT2
    Root Directory
    Data
Logical Disk . . . .
```

Figure 3.2 *Hard disk contents.*

The Disk Boot Program

The boot sector is the first sector of the boot drive and contains the boot program (disk bootstrap loader). The disk bootstrap loader is copied to memory at `0:7C00h` by the ROM bootstrap loader. Control is then transferred to the disk boot program. Hard disks normally have two different boot sectors and two boot programs, a master boot sector that is the first physical sector of the disk and a normal boot sector that is the first logical sector of the disk's bootable partition. The master boot sector contains the partition table for the hard disk. The partition table describes the logical disks comprising the physical drive. Floppy disks cannot have multiple partitions, so the first physical sector is always the boot sector.

Master Boot Sector

The master boot sector of a hard disk contains its partition table and a boot program. The boot program mimics the ROM bootstrap loader in loading the boot sector of the bootable logical disk and transferring control to the second boot program. The master boot sector (or master record, master boot program) resides on the first absolute or physical sector of a hard drive. The boot program used by the bootstrap loader resides in the first logical sector of the bootable partition. The master boot program is accompanied in the first physical sector by the partition table, consisting of four entries which occupy bytes `446` through `508` (`1be`—`1fe`), with the final signature bytes, `55h`, `0AAh`.

The master boot program performs the following tasks:

- Copies itself (the master boot program) to another location, since the ROM bootstrap loaded the master record at `0:7C00h` and standard boot sectors expect to be loaded there as well.
- Locates a bootable partition (logical disk) from the partition table (and may check for correct partition table structure).
- Uses BIOS `INT 13h`, to read a sector to load the boot record from the first sector of the bootable disk.
- Checks for the signature bytes `55h`, `0AAh` at the end of the second boot sector.
- Transfers control to the newly loaded disk bootstrap program. If the boot attempt fails, the master boot program may reboot or execute `INT 18h`, boot failure.

Partition Table

Each entry of the partition table describes the characteristics of one logical disk drive. Partition table entries are organized as described in Table 3.1.

Disk Bootstrap Loader

The disk bootstrap loader, the boot program from the bootable partition of a hard drive or the boot sector of a floppy, loads the system file `IO.SYS` and may load the DOS kernel, `MSDOS.SYS`. If the boot program cannot load these files, the loader normally prompts for a system disk and reboots. If the system files are loaded correctly, control is transferred to the BIOS initialization routines of `IO.SYS`. The bootstrap loader expects `IO.SYS` to be the first entry in the root directory and the first file in the disk's data area, stored in sequential sectors. The sectors of `IO.SYS` are copied into RAM at location `0:700h`.

Boot Sector Organization

The first sector of every "logical" disk is the boot sector (sometimes called the boot record). The term "boot record" is best used for the data

Offset Hex	**Example Values***	**Field Size**	**Description**
0	80h	1	boot indication 0-non-bootable, 80-bootable
1	1	1	first head
2	1	6 bits	first sector (relative to current cylinder) (low 6 bits only, upper bits extend cylinder byte)
3	0	10 bits	first cylinder
4	4	1	system ID (0-unknown, 1-DOS/12-bit FATs, 40DOS/16-bit FATs)
5	4	1	last head
6	11h	6 bits	last sector (relative to current cylinder) (low 6 bits only, upper bits extend cylinder byte)
7	2DBh (731 decimal)	10 bits	last cylinder
8	11h	4	first sector (absolute)
C	FBF2h (62,203 decimal)	4	# sectors

* Example values are drawn from the partition table of a Seagate 4038 disk with a single, bootable, DOS v3.3 partition.

Table 3.1 *Partition table.*

in the boot sector, distinguished from the executable code, which is called the boot program. Table 3.2 illustrates the boot sector organization.

Offset hex/dec	**Description**	**Field Size (bytes)**	**Example***
Jump Instruction:			
0/0	Jump to boot program	3	JMP 3Eh
Boot Record Structure:			
3/3	System ID	8	“IBM 3.3”
BIOS Parameter Block (BPB):			
B/11	bytes/sector	2	100h (512)
D/13	sectors/cluster	1	4
E/14	# reserved sectors from 0	2	1
10/16	# of FATs	1	2
11/17	max # root directory entries	2	200h (512)
13/19	Total sectors on disk	2	PF2FBh (62,203)
	(in DOS v3+, an entry of 0 indicates the use of a double word for total sector count, see offset 20h		
15/21	format ID (media descriptor)	1	0F8h
16/22	sectors/FAT	2	3Dh (61)
18/24	sectors/track	2	11h (17)
1A/26	# of sides (heads)	2	5
1C/28	# hidden sectors	2	11h (17)
1E/30	# hidden sectors	2	80h
	(in DOS v4, double word entry for hidden sectors, these are the upper two bytes with previous word as lower two bytes)		
20/32	4 byte total sectors/disk	4	DOS v3 0
24/36	physical drive #	1	DOS v4 0
25/37	reserved	1	0
26/38	extended boot record signature	1	0
27/39	volume serial #	4	0
2B/43	volume label	11	0
36/54	reserved	8	0

3E.62	Boot Program resumes		cli
			xor ax,ax
			mov sx,ax
			mov sp,7C00h
			...

	Program Data		“Non-System dis...” ...
1FE/510	Signature	2	55AAh
end of Boot Sector			
*Example values are drawn from the boot sector of a Seagate 4038 disk, DOS v3.3.			

Table 3.2 *Boot sector.*

Memory Organization

8088, Real-mode 80x86 memory

ROM:	
`A0000-BFFFF`	Video Adapter ROM and RAM Space
`C0000-DFFFF`	Adapter Board ROM and RAM Space (e.g., disk controller)
`E0000-FFFFF`	ROM BIOS (may include Basic Interpreter in ROM) POST BIOS
`0F000:E6F2h`	Bootstrap Loader
RAM:	
`00000-003FF`	Interrupt Vector Table
`00400-004FF`	ROM BIOS data area
`00472`	Initialization Flag

80x86 Extended Memory

ROM:	
`FE0000-FFFFFF`	ROM BIOS
RAM:	
`100000-FDFFFF`	Available

Important Memory Locations

`F000:FFF0`	Initial CS:IP contents at power-on, system startup routine
`F000:0000 - F000:FFFF`	ROM BIOS
`0000:0000 - 0000:03FF`	Interrupt table
`0000:0400 - 0000:05FF`	BIOS tables and data area
`0000:0472`	reset flag
`0000:7C00`	ROM bootstrap loader destination for boot sector

Important ROM BIOS Interrupts

INT 10, *service routine* — 0F000:F065h

Video Services:

Function code in register ah.

Function ah = 0Eh, write character in teletype mode
al = character to write
bh = display page
bl = foreground color
returns nothing

INT 13, *service routine* — 0F000:E3FEh

Disk Services:

Function code in register ah.

Function ah = 0, reset disk system
returns carry flag set on error
Function ah = 2, read disk sectors
al = # sectors to read
ch = cylinder/track number
hard disks use upper two bits
of cl as high order extension
cl = sector number, 0 - 63, bits 0 through 5 only
DH = head/side number, 0 - ?
DL = drive number, 0 = A, 1 = B, 80 = C, 81 = D, ...
ES:BX = destination pointer
returns carry flag set on error
ah = error code
al = # sectors transferred

INT 16, *service routine* — 0F000:E84Eh

Keyboard Services:

Function code in register ah.

Function ah = 0, read keyboard input
returns ah = keyboard scan code
al = ASCII code

`INT 18`, *service routine* — `0F000:1C90h`

ROM Basic:

Triggered during boot process if a bootable disk cannot be found. The IBM BASICA interpreter uses routines stored in association with the IBM version of *INT 18h*. Compatibles do not include the ROM Basic and so cannot execute IBM BASICA.

`INT 19h`, *service routine* — `0F000:E6F2h`

Bootstrap Loader (System Warm Boot)

May reset portions of the interrupt vector table in low memory. Initiates boot sequence. Reloads and Starts Operating system without losing RAM contents (except where overwritten as part of loading and starting OS).

Search A: and C: for bootable disk. *DL* register is set to drive number of bootable disk. Load first sector (track 0, sector 1) into memory at *007C:0000h*. If no boot sector found, executes *INT 18h*.

NOT identical to a "power-off restart" (resets entire system & POST) NOR Ctrl-Alt-Del — (resets machine state and memory allocations). These are initiated by branching to *0FFFF:0000h*, the system power-on location. The contents of *40:72h* dictate the thoroughness of *POST* and initialization.

Glossary

BIOS — Basic Input/Output System

BIOS Parameter Block (BPB) — a data structure that describes the attributes of a block storage device, such as a disk. The BPB is part of the boot record and often is used synonymously with the entire data area of the boot sector. A table of BPBs is maintained to control access to block devices.

boot (or bootstrap) — to begin incrementally, to "pull yourself up by your bootstraps," refers to process by which a computer system starts operation from a very small program stored in ROM (non-volatile memory) and gradually proceeds to load and execute more complex programs, initializing and testing the system along the way, until the entire operating system and all device drivers are executing or ready to execute. Searches disks for system files to load.

boot program — executable portion of boot sector. Typically loads and executes *IO.SYS*.

boot record — term commonly used for entire boot sector, perhaps best used for the data stored in a boot sector, excluding the executable code.

boot sector (boot record) — single sector specifying details of disk and DOS, contains program that loads operating system kernel (*IO.SYS*) into memory and transfers control to operating system initialization routine. The first sector of every *logical* disk is the boot sector

cluster — a group of sectors. DOS allocates disk space in terms of clusters, with FATs recording chains of clusters comprising files. Clusters range in size from two to eight sectors, depending on the particular hardware, though two sectors per cluster is common.

FAT — file allocation table — indicates chains of clusters allocated to files on a disk. DOS normally maintains two copies of the FAT. FATs may either have 12- or 16-bit entries (and hence disks may have a maximum of either 4098 or 65,536 clusters).

master boot sector (master boot program, master record) — first physical sector of hard disk, contains the partition table and a bootstrap loader that loads the boot record of the bootable partition

partition — a region of a hard disk, used as a logical drive. Hard disks may have one to four partitions, which may be assigned to different operating systems. At most one partition can be bootable.

partition table — table contained in master boot program that specifies partition sizes, locations and uses. Indicates whether a particular partition is bootable.
`POST` — `Power-on Self Test`. Sequence of test programs executed from ROM when system is reset or turned on (cold booted).
ROM — Read Only Memory

System Files

`COMMAND.COM` — command interpreter; loads and executes programs
`IO.SYS (IBMBIO.COM)` — first file loaded during boot, extends ROM BIOS, must be first file in root directory, first file on the disk, stored in contiguous clusters in order
`MSDOS.SYS (IBMDOS.COM)` — second file loaded during boot, loaded by `IO.SYS`, contains DOS kernel, must be second file in root directory, Before MSDOS v3.0, `MSDOS.SYS` was required to be in ordered, contiguous clusters following `IO.SYS`.
`CONFIG.SYS` — system buffer and device driver configuration commands. Loaded by `IO.SYS`.
`AUTOEXEC.BAT` — batch file loaded and executed by `COMMAND.COM` as last task of system boot process.
`SYS.COM` — program that stores `IO.SYS` and `MSDOS.SYS` on floppy (along with boot sector) to make the floppy disk bootable.

Important References

Phoenix Technologies Ltd, *System BIOS for IBM PC/XT/AT Computers and Compatibles,* Addison Wesley, 1989.

An excellent reference on BIOS functions, tables and data areas. No details on BIOS code, though.

Terry Dettman, Jim Kyle, *DOS Programmers Reference*, 2nd Edition, QUE, 1989.

Every programmer should have this one.

Ray Duncan, *Advanced MSDOS*, Microsoft Press, 1986.

Dated, but excellent discussion of boot process. A second edition (1988) includes v4 of DOS.

Paul Somerson, *DOS Power Tools*, Bantam, 1988.

Learn to use `DEBUG`. There is now a second edition of this book.

IBM Technical Reference, Personal Computer Hardware Reference Library, 1981.

The old ones with the ROM BIOS listings! Dated, but there's nothing like commented listings for figuring out the details.

Listing 3.1 BOOT360.ASM

```
.MODEL tiny
.radix 16

high_bios segment at 0F000h
    assume cs:high_bios
    org 0FFF0h
restart label far
high_bios ends

low_bios segment at 40h
    assume es:low_bios
    org 72h
reset_flag label word
low_bios ends

warm     equ 1234h
cold     equ 7F7Fh ; alternative values to store in reset flag
mem_sav  equ 4321h ; affect extent of system restart
boot_loc equ 7C00h

MAIN    segment
    assume cs:main, ds:main, ss:main
    jmp short boot_program
    nop
BIOS_Parameter_Block:
    db 'BOOT skg'
    dw 200h        ; bytes/sector
    db 2           ; sectors/cluster (often 4 for HD)
    dw 1           ; reserved sectors (from sector 0)
    db 2           ; typically 2 FATs
    dw 70h         ; # entries in root directory (200h for HD)
    dw 2D0h        ; sectors/disk
    db 0FFh        ; format ID/media descriptor/disk type
    dw 1           ; sectors/FAT
    dw 9           ; sectors/track
    dw 2           ; sides (heads)/disk
    db 22h dup (0) ; reserved sectors
```

Listing 3.1 *continued*

```
boot_program:
    mov ax,low_bios         ; set system reset flag
                            ; (0040:7200) to warm boot
    mov es,ax
    mov ax,warm
    assume es:low_bios
    mov word ptr [reset_flag],ax

    mov ax,cs               ; setup segment registers
    mov ds,ax
    mov ss,ax
    mov sp,boot_loc
    mov si,boot_loc + offset Message

    call display_string
    jmp end_program

display_string:             ; ds:si points to string
    lodsb
    or al,al
    jz end_string_loop
    mov ah,0Eh
    mov bx,7
    int 10h
    jmp display_string
end_string_loop:
    ret

end_program:
    xor ax,ax
    int 16h                 ; wait for any key
    jmp far ptr restart
;       program ends with the preceding jump to restart
;       could use int 19h rather than jmp far ptr restart
;       Avoids some system initialization and POST

CR      equ 0Dh
LF      equ 0Ah

Message db CR,LF,'< !YOUR MESSAGE GOES HERE! >'
    db CR,LF,'This is not a bootable disk!'
    db CR,LF,'Remove disk and press any key.'
    db CR,LF,0

    org 1FEh
Signature:
    db 55h,0AAh

MAIN    ends
    end

; End of File
```

Listing 3.2 *Floppy BIOS parameter blocks.*

```
; for 720Kb floppy
BIOS_Parameter_Block:
    db 'BOOT skg'
    dw 200h        ; bytes/sector
    db 2           ; sectors/cluster (often 4 for HD)
    dw 1           ; reserved sectors (from sector 0)
    db 2           ; typically 2 FATs
    dw 70h         ; # entries in root directory (200h for HD)
    dw 5A0h        ; sectors/disk
    db 0F9h        ; format ID/media descriptor/disk type
    dw 3           ; sectors/FAT
    dw 9           ; sectors/track
    dw 2           ; sides (heads)/disk
    dd 0           ; number of hidden sectors
    dd 0           ; when sectors/disk field = 0, then this field specifies sectors/disk

; for 1.2Mb 5.25" floppy:
BIOS_Parameter_Block:
    db 'BOOT skg'
    dw 200h
    db 1
    dw 1
    db 2
    dw 0E0h
    dw 960h
    db 0F9h
    dw 7
    dw 0Fh
    dw 2
    dd 0
    dd 0

; for 1.44Mb 3.5" floppy:
BIOS_Parameter_Block:
    db 'BOOT skg'
    dw 200h
    db 1
    dw 1
    db 2
    dw 0E0h
    dw 0B40h
    db 0F0h
    dw 9
    dw 12h
    dw 2
    dd 0
    dd 0

; End of File
```

Listing 3.3 BOOT4038.ASM

```
.MODEL tiny
.radix 16

high_bios segment at 0F000h
    assume cs:high_bios
    org 0FFF0h
restart label far
high_bios ends

low_bios segment at 40h
    assume es:low_bios
    org 72h
reset_flag label word
low_bios ends

warm     equ 1234h
cold     equ 7F7Fh ; alternative values to store in reset flag
mem_sav  equ 4321h ; affect extent of system restart
boot_loc equ 7C00h

MAIN    segment
    assume cs:main, ds:main, ss:main
    jmp short boot_program
    nop
BIOS_Parameter_Block:
    db 'BOOT skg'
    dw 200h          ; bytes/sector
    db 4             ; sectors/cluster (often 4 for HD)
    dw 1             ; reserved sectors (from sector 0)
    db 2             ; typically 2 FATs
    dw 200h          ; # entries in root directory (70h for FDD)
    dw 0F2FBh        ; sectors/disk
    db 0F8h          ; format ID/media descriptor/disk type
    dw 3Dh           ; sectors/FAT
    dw 11h           ; sectors/track
    dw 5             ; sides (heads)/disk
    dw 11h           ; reserved sectors
    dw 80h           ;
    db 0Fh dup (0)   ; reserved
    db 12h
    db 4 dup (0)
    db 1
    db 9 dup (0)
```

Listing 3.3 *continued*

```
boot_program:
    mov ax,low_bios ; set system reset flag (0040:7200) to warm boot
    mov es,ax
    mov ax,warm
    assume es:low_bios
    mov word ptr [reset_flag],ax

    mov ax,cs       ; setup segment registers
    mov ds,ax
    mov ss,ax
    mov sp,boot_loc
    mov si,boot_loc + offset Message

    call display_string
    jmp end_program

display_string:         ; ds:si points to string
    lodsb
    or al,al
    jz end_string_loop
    mov ah,0Eh
    mov bx,7
    int 10h
    jmp display_string
end_string_loop:
    ret

end_program:
    xor ax,ax
    int 16h         ; wait for any key
    jmp far ptr restart
;       program ends with the preceding jump to restart
;       could use int 19h rather than jmp far ptr restart
;       Avoids some system initialization and POST

CR      equ 0Dh
LF      equ 0Ah

Message db CR,LF,'< !YOUR MESSAGE GOES HERE! >'
    db CR,LF,'This is not a bootable disk!'
    db CR,LF,'Insert bootable floppy disk and press any key.'
    db CR,LF,0

    org 1FEh
Signature:
    db 55h,0AAh

MAIN    ends

    end

; End of File
```

Listing 3.4 BOOTPASS.ASM

```
.MODEL tiny
.radix 16

high_bios segment at 0F000h
    assume cs:high_bios
    org 0FFF0h
restart label far
high_bios ends

low_bios segment at 40h
    assume es:low_bios
    org 72h
reset_flag label word
low_bios ends

warm     equ 1234h
cold     equ 7F7Fh       ; alternative values to store in reset flag
mem_sav  equ 4321h       ; affect extent of system restart
boot_loc equ 7C00h

MAIN    segment
    assume cs:main, ds:main, ss:main
    jmp pass_check

BIOS_Parameter_Block:
    db 'BOOT skg'
    dw 200h          ; bytes/sector
    db 4             ; sectors/cluster (often 4 for HD)
    dw 1             ; reserved sectors (from sector 0)
    db 2             ; typically 2 FATs
    dw 200h          ; # entries in root directory (70h for FDD)
    dw 0F2FBh        ; sectors/disk
    db 0F8h          ; format ID/media descriptor/disk type
    dw 3Dh           ; sectors/FAT
    dw 11h           ; sectors/track
    dw 5             ; sides (heads)/disk
    dw 11h           ; reserved sectors
    dw 80h           ;
    db 0Fh dup (0)   ; reserved
    db 12h
    db 4 dup (0)
    db 1
    db 1 dup (0)

boot_program:
    mov ax,low_bios ; set system reset flag (0040:7200)
                    ; to warm boot
    mov es,ax
    mov ax,warm
    assume es:low_bios
    mov word ptr [reset_flag],ax
```

Listing 3.4 *continued*

```
    mov ax,cs         ; setup segment registers
    mov ds,ax
    mov ss,ax
    mov sp,boot_loc
    mov si,boot_loc + offset Message

    call display_string
    jmp end_program

display_string:      ; ds:si points to string
    lodsb
    or al,al
    jz end_string_loop
    mov ah,0Eh
    mov bx,7
    int 10h
    jmp display_string
end_string_loop:
    ret

end_program:
    xor ax,ax
    int 16h           ; wait for any key
    jmp far ptr restart
;       program ends with the preceding jump to restart
;       could use int 19h rather than jmp far ptr restart
;       Avoids some system initialization and POST

CR      equ 0Dh
LF      equ 0Ah

Message db CR,LF,'< !YOUR MESSAGE GOES HERE! >'
    db CR,LF,'Access restricted. Turn off computer.'
    db CR,LF,'Contact system owner.'
    db CR,LF,0

    org 1EEh
pass_check:
    xor ax,ax
    int 16
    cmp ax,011Bh
    jz short ok
    jmp end_program
ok:
    jmp boot_program
    org 1FEh
Signature:
    db 55h,0AAh

MAIN    ends

    end

; End of File
```

Listing 3.5 BOOTRUN.ASM

```
.MODEL tiny
.radix 16

high_bios segment at 0F000h
    assume cs:high_bios
    org 0FFF0h
restart label far
high_bios ends

low_bios segment at 40h
    assume es:low_bios
    org 72h
reset_flag label word
low_bios ends

bootsys segment at 7E0h
    assume cs:bootsys,ds:bootsys
    org 100h
run_bootsys label far
bootsys ends

warm     equ 1234h
cold     equ 7F7Fh ; alternative values to store in reset flag
mem_sav  equ 4321h ; affect extent of system restart
boot_loc equ 7C00h
bootsys_loc equ 7E00h
MAIN    segment
    assume cs:main, ds:main, ss:main
    jmp short boot_program
    nop
BIOS_Parameter_Block:
    db 'BOOT skg'
    dw 200h          ; bytes/sector
    db 2             ; sectors/cluster (often 4 for HD)
    dw 1             ; reserved sectors (from sector 0)
    db 2             ; typically 2 FATs
    dw 70h           ; # entries in root directory (200h for HD)
    dw 2D0h          ; sectors/disk
    db 0FFh          ; format ID/media descriptor/disk type
    dw 1             ; sectors/FAT
    dw 9             ; sectors/track
    dw 2             ; sides (heads)/disk
    db 22h dup (0)   ; reserved sectors
```

Listing 3.5 *continued*

```
boot_program:
    mov ax,cs        ; setup segment registers ax=cs=0
    mov ds,ax
    mov es,ax
    mov ss,ax
    mov sp,boot_loc

    int 13           ; reset disk system, (ax <-- 0 above)
    jc errors

    mov ax,0201h     ; read first sector of root directory
    mov bx,bootsys_loc      ; into area 0000:7E00h +
    mov cx,0004h     ; track 0, sector 4 - root directory
    mov dx,0000h     ; side 0, disk 0
    int 13
    jc errors

    mov cx,0bh       ; check first program name is 'BOOTSYS BIN'
    mov di,bx
    mov si,boot_loc + offset bootsys_name
    repz
    cmpsb
    jcxz load_sys
    jmp errors

load_sys:
    mov ax,0206h         ; read first six data sectors
    mov bx,bootsys_loc   ; into area 0000:7E00h +
    mov cx,0003h         ; track 0, sector 3 - start data area
    mov dx,0100h         ; side 1, disk 0
    int 13
    jc errors

    jmp far ptr run_bootsys ; execute bootsys.bin

errors:
    mov ax,low_bios      ; set system reset flag (0040:7200) to warm boot
    mov es,ax
    mov ax,warm
    assume es:low_bios
    mov word ptr [reset_flag],ax

    mov si,boot_loc + offset Message

    call display_string
    jmp end_program
```

Listing 3.5 *continued*

```
display_string:          ; ds:si points to string
    lodsb
    or al,al
    jz end_string_loop
    mov ah,0Eh
    mov bx,7
    int 10h
    jmp display_string
end_string_loop:
    ret

end_program:
    xor ax,ax
    int 16h          ; wait for any key
    jmp far ptr restart
;       program ends with the preceding jump to restart
;       could use int 19h rather than jmp far ptr restart
;       Avoids some system initialization and POST

CR      equ 0Dh
LF      equ 0Ah

bootsys_name db 'BOOTSYS BIN',0

Message db CR,LF,'File BOOTSYS.BIN not found.'
    db CR,LF,'This is not a bootable disk!'
    db CR,LF,'Remove disk and press any key.'
    db CR,LF,0

    org 1FEh
Signature:
    db 55h,0AAh

MAIN    ends

    end

; End of File
```

Listing 3.6 BOOTSYS.ASM

```
.MODEL tiny
.radix 16

high_bios segment at 0F000h
    assume cs:high_bios
    org 0FFF0h
restart label far
high_bios ends

low_bios segment at 40h
    assume es:low_bios
    org 72h
reset_flag label word
low_bios ends

warm     equ 1234h

MAIN    segment
    assume cs:main, ds:main, ss:main

    org 100h
run_bootsys:
    mov ax,cs           ; setup segment registers
    mov ds,ax
    mov ss,ax
    mov sp,offset stack_area

; code for desired boot system actions should be inserted here.
; currently, displays possibilities and reboots

    mov si,offset Actions
    call display_string
    jmp end_program
```

Listing 3.6 *continued*

```
display_string:         ; ds:si points to string
    lodsb
    or al,al
    jz end_string_loop
    mov ah,0Eh
    mov bx,7
    int 10h
    jmp display_string
end_string_loop:
    ret

end_program:
    mov ax,low_bios     ; set system reset flag (0040:7200)
                        ; to warm boot
    mov es,ax
    mov ax,warm
    assume es:low_bios
    mov word ptr [reset_flag],ax
    mov si,offset End_Message

    call display_string

    xor ax,ax
    int 16h         ; wait for any key
    jmp far ptr restart
;       program ends with the preceding jump to restart
;       could use int 19h rather than jmp far ptr restart
;       Avoids some system initialization and POST

CR      equ 0Dh
LF      equ 0Ah

Actions db CR,LF,'- Diagnose System.'
    db CR,LF,'- Scan for Virus.'
    db CR,LF,'- Implement Alternative Operating System'
    db CR,LF,0

End_Message db CR,LF,'BOOTSYS.BIN has finished executing.'
    db CR,LF,'Remove disk and press any key to reboot.'
    db CR,LF,0

    org 3FEh
stack_area dw 0

MAIN    ends

    end

; End of File
```

Chapter 4

Converting a Microsoft C Program into a TSR

Michael J. Young

A memory resident program, or Terminate-and-Stay-Resident (TSR) application), is one that permanently establishes itself in memory and then returns control to MS-DOS so that the user can run other programs. The TSR remains dormant in the background until the user presses a designated hot key, whereupon it springs to life, temporarily suspending the current application and providing instant access to the services it offers.

The module of functions presented in this chapter allows you to convert a normal Microsoft C program into a TSR through a single function call. You can specify the hot key that will activate your TSR, and the C function in your program that will receive immediate control when the hot key is pressed. You can also specify a unique code to prevent the same resident program from being installed more than once. All the details of installing and activating the program as a TSR are handled invisibly by the functions in this module. The module also includes a function for removing the TSR from memory.

[Reprinted from *Systems Programming in Microsoft C*, by Michael Young, by permission of SYBEX, Inc. ISBN Number 0-89588-570-0, copyright 1989, SYBEX, Inc. All rights reserved.]

The TSR functions written in C are listed at the end of this chapter in Listing 4.1, and the functions written in assembly language are listed in Listing 4.2. The header file that you must include in your C program to call these functions is given in Listing 4.3.

Writing a TSR under MS-DOS is a complex task. The primary challenge is to design a resident program that will peacefully coexist with the foreground program, with any other TSRs that are currently installed, and with the operating system. It is especially important that a TSR written in the C language be able to freely call MS-DOS functions, since many of the standard library routines make use of these services. This chapter presents one of the many approaches that can be used to achieve these design goals.

There are three functions in the TSR module that can be called directly by your program: `TsrInstall`, `TsrInDos`, and `TsrRemove`. The chapter begins by describing how to call these three functions and presents an example program that demonstrates their use. The chapter then discusses how the functions work.

The TSR functions not only provide a useful addition to your collection of software tools, but they also furnish an excellent demonstration of the advanced features of Microsoft C. Normally it would be quite difficult to write memory resident interrupt handlers in the C language; however, using Microsoft C language extensions such as interrupt functions (provided with versions 5.0 and later), it was possible to write the entire module in C, with the exception of two short assembly language subroutines. The functions are compatible with the Microsoft optimizing compiler versions 5.0 through 6.0, and with all versions of QuickC.

Using `TsrInstall`

The function `TsrInstall` has the prototype

```
int TsrInstall (void (*FPtr) (void),
                int hot key, unsigned long Code)
```

and performs the following primary tasks:

- It checks to make sure that the program has not already been installed in memory. Each program that is installed by this routine should be identified with a unique value in the parameter `Code`.

- It stores all values that will be required by the interrupt handlers after the TSR is installed.
- It installs the appropriate interrupt handlers so that when the user presses the shift-key combination specified by the `hot key` parameter, the function given by the `FPtr` parameter receives immediate control.
- It terminates the C program, leaving the entire block of code and data resident in memory.

When you develop a memory resident program using the TSR module, your program should consist of two parts. The first part is the `main` function (plus any functions called by `main`). The function `main` receives control when the user types the name of the program on the command line and serves to install the TSR. This function should perform any required initialization tasks, print any desired messages to the user, and finally call `TsrInstall` to install the program as a TSR. After the call to `TsrInstall`, control returns to the DOS command line just as if the program had exited normally; however, the program's code and data are not released, but rather remain resident in memory.

The second part of your memory resident program is the portion of the code that is subsequently activated whenever the user presses the designated hot key. This part of the program consists of the TSR entry routine — the function that receives initial control when the hot key is pressed — plus any subroutines called by this function.

The first parameter passed to `TsrInstall` (that is, `FPtr`) should be assigned the address of the TSR entry routine. The second parameter (`hot key`) gives the shift-key combination that is used to activate the program. Note that this parameter does not specify either an ASCII code or an extended keyboard code, but rather some combination of shift keys. The desired combination may be specified by joining two or more of the following constant identifiers with the C `|` operator (these constants are defined in the header file `TSR.H` in Listing 4.3):

Identifier	**Shift Key**
`RIGHTSHIFT`	Right Shift key
`LEFTSHIFT`	Left Shift key
`CONTROL`	Ctrl key
`ALT`	Alt key
`SCROLLLOCK`	Scroll Lock key
`NUMLOCK`	Num Lock key
`CAPSLOCK`	Caps Lock key
`INSERT`	Ins key

For example, the combination of the `Alt-key` and `Left-Shift` key could be specified by the following expression:

```
ALT | LEFTSHIFT
```

The parameter `Code` is of type `unsigned long`. It should range between the values `0x00000001` and `0xffffffff` and be unique for each separate program you convert to a TSR. Once a TSR is installed, no other program having the same ID number may subsequently be installed. By means of this parameter, `TsrInstall` avoids loading redundant copies of the same program.

Since `TsrInstall` terminates the current program, it should obviously never return. If the function does return, an error has occurred and the program has not been installed. In this case, one of the following error codes is passed back to the calling program:

- `INSTALLED`: This error code indicates that a TSR bearing the same identification code has already been installed.
- `NOINT`: This value means that there are no free "user" interrupts available. The interrupt vectors in the range from `60h` to `67h` are designated as available for user programs. The first of these vectors that is `free` (that is, containing `0`) is used to store the identifying code (the parameter `Code`) assigned to the TSR. If all of these vectors are occupied, `TsrInstall` returns `NOINT`.
- `WRONGDOS`: The TSR functions require DOS v2.0 or later. If the version is prior to v2.0, this error code is returned. (Note that the TSR can be successfully installed in the DOS-compatible environment of OS/2, which returns a version number of 10.0 or higher.)

- `ERROR`: This general code is returned if the installation procedure failed for some unknown reason.

In most respects, a program destined to become a TSR can be written like any other C program. However, there are five additional guidelines that you should follow when developing your memory resident program.

First, your program — and all the C modules that are linked with it — must be compiled using the small or medium memory model (both of these models place all data within a single segment and use near data pointers by default). You may not compile the program using either the compact, large, or huge model. As you will see, the code in the TSR module (specifically, `TSR.C`) uses techniques that require the small or medium memory model.

Second, if your program writes to the screen or alters the cursor, you must carefully save and restore the complete video state of the interrupted program. The functions in the TSR module automatically save and restore almost all of the machine state of the suspended program; the task of preserving the video state, however, is left to the application program. Note that you can use the routines `ScrSaveBlock` and `ScrRestoreBlock` (defined in Listing 4.4) to save and restore data from text mode screens.

Third, a resident program should not attempt to allocate memory from DOS. A transient application (that is, a program that does not remain resident in memory) can normally obtain additional memory blocks by dynamically allocating them from DOS. A TSR, however, should restrict its use of memory to the block explicitly reserved for the program when it terminated and remained resident. Consequently, a TSR should not invoke the MS-DOS services for allocating, releasing, or changing the size of memory blocks (namely, services `48h`, `49h`, and `4Ah`, accessed through interrupt `21h`); likewise, it should not call any of the functions provided by Microsoft C that rely upon these DOS services (namely, `_dos_allocmem`, `_dos_freemem`, and `_dos_setblock`). Note, however, that a TSR can call the standard C memory allocation functions, such as `malloc` and `calloc`, since these functions obtain memory from within the block reserved at the time the program became resident. (However, as you will see in the next section, the TSR module places a fixed upper bound on the amount of memory that can be dynamically allocated through functions such as `malloc`.)

A corollary to the third guideline is that a TSR must not attempt to execute a child process, since doing so entails allocation of additional memory blocks. Consequently, a TSR must not call the DOS service for executing a child process (namely, function `4Bh` of interrupt `21h`); nor should it call any of the C library functions for starting new processes (namely, the `exec...` and `spawn...` families of functions).

Fourth, the TSR should not use floating point numbers. Specifically, it should not declare variables of type `float` or `double`, and it should not call functions in the C math library (which are those declared in the header file `MATH.H`). The inclusion of floating point code causes the C startup code to set certain interrupt vectors; floating point operations require that these interrupt vectors remain intact. As long as these vectors remain unaltered as various programs run in the foreground, the TSR can successfully execute floating point instructions. However, if a foreground application alters one or more of the vectors, any floating point calculations performed by the TSR will fail. Therefore, a robust TSR must avoid floating point operations performed by the math library.

The fifth and final guideline is that your TSR must always return from the function that was initially activated (the function specified by the `FPtr` parameter). You must not use the `exit` function to terminate the program. The `exit` function is designed to terminate a non-resident program and return to the parent process (usually the MS-DOS command interpreter).

Note that when your TSR entry function receives control in response to the user pressing the hot key, the TSR module has temporarily disabled any control-break or critical error handlers that may have been installed by the suspended foreground application. (Specifically, during the execution of your TSR code, interrupt vectors `1Bh`, `23h`, and `24h` point to routines that simply return without performing any action; these vectors are restored to their former values before control returns to the suspended application.) If your TSR program needs to handle control-break keys or critical-error conditions, you can install appropriate handlers. However, you do not need to restore the former values for these vectors since this task is performed automatically by the TSR module code that receives control when your TSR issues its final return statement. (Editor's note: See Chapters 10 and 11 for *Ctrl-C* and critical error handlers.)

Note also that even if the TSR is properly installed, the program may not always receive immediate control when the hot key is pressed. There are times when it is not safe to interrupt the current process (such as when

DOS or the BIOS are engaged in disk activity). The TSR module senses these conditions and simply ignores the hot key until it is safe to trigger the resident program.

A note to QuickC users: You should not attempt to load a TSR from within the QuickC integrated environment (the TSR should be loaded directly from the DOS command line so that it is given the lowest possible position in memory). Therefore, if you prepare the program within QuickC v1.0, you should select the `Exe` item in the `Output Options` column of the `Compile` menu. This option will produce a free-standing `.EXE` file that can be run from DOS. QuickC v2.0 will automatically produce such a file.

Using `TsrInDos`

The function `TsrInDos` has the prototype

```
int TsrInDos (void)
```

This function returns a non-zero value if an MS-DOS service was active when your TSR received control, and it returns `0` if MS-DOS was not active. With this function you may determine which DOS functions your program may safely call. Specifically, if `TsrInDos` returns a value of `0`, you may directly call almost any MS-DOS service or almost any C library function that invokes MS-DOS (exceptions are noted later in this section). However, if `TsrInDos` returns a non-zero value, then you must not use DOS functions `01h` through `0Ch`. This set of forbidden functions manage input and output to the basic character devices (console, printer, and serial port). The following C library functions, which are defined in the `CONIO.H` header file, employ these services: `cgets`, `cputs`, `cprintf`, `cscanf`, `getch`, `getche`, `kbhit`, `putch`, and `ungetch`.

Fortunately, it is almost always possible to obtain the services offered by the forbidden set of procedures through the low-level C library functions provided by Microsoft (notably `_bios_keybrd` and display functions such as `_outtext`), or by using some of the video functions presented in Listing 4.4. Either you can call `TsrInDos` before using any of the restricted functions, or you can simply eliminate the restricted functions from your TSR program. Later in this chapter I'll explain why you should not call this set of MS-DOS services.

Using TsrRemove

The function *TsrRemove* has the prototype

```
int TsrRemove (void)
```

The memory resident portion of your program can call *TsrRemove* to remove itself from memory. It should call this function immediately before returning control to the foreground program. *TsrRemove* releases the memory occupied by the TSR and restores all interrupt vectors to their former values.

If successful, *TsrRemove* returns the value *NOERROR* (which equals *0*; error constants are defined in *TSR.H*, Listing 4.3). At certain times, however, it is not safe for the TSR to remove itself from memory (specifically, when the foreground application or a subsequently loaded TSR has hooked one or more of the interrupt vectors used by the module). In this case, *TsrRemove* returns the value *CANTREMOVE*. It may be possible to remove the TSR after terminating the foreground application, or after removing any other TSRs that were installed after your program.

An Example Program

The program in Listing 4.4 demonstrates the use of *TsrInstall*, *TsrInDos*, and *TsrRemove*. This program consists of two functions: *main* and *Test*. The function *main* calls *TsrInstall* to convert the program to a TSR, specifying *Test* as the function to receive control when the user presses the hot key. The selected hot key is *Ctrl-Left Shift*, and the identifying code is *0x11111111*. If another TSR is developed that might be installed at the same time as this program, a different hot key combination and identifying code should be chosen. If *TsrInstall* is successful, it will never return. However, if an error occurs in the installation process, control returns to *main*, which then prints the appropriate error message and terminates with an *ERRORLEVEL* setting of *1*.

If *TsrInstall* is successful, control returns to DOS. Subsequently, each time the user presses *Ctrl-Left Shift* (and it is safe to interrupt the current process), the function *Test* receives control. This function saves and restores the current screen contents using *ScrSaveBlock* and *ScrRestoreBlock*, displays a window, and pauses for user input by calling the Microsoft C function `_bios_keybrd`. Note that this program, written to provide a simple example, will work properly only if the system is in a text mode. Since `Test` does not alter the cursor, it does not need to preserve the cursor parameters.

If the user presses r or R, `Test` calls `TsrRemove` to remove the TSR from memory. If the user presses any other key, the function simply continues.

Listing 4.5 provides a `MAKE` file for preparing both the TSR function module and the example program.

How the TSR Functions Work

The General Strategy

Before discussing the details of the individual functions in the TSR module, this section provides an overview of the basic tasks required to install and activate a memory resident program, and the general strategies that the TSR module uses to accomplish these tasks.

The most fundamental task required to install a TSR is to load the program into a safe area of memory that will not be overwritten by subsequent applications. Fortunately, MS-DOS provides a service that allows a program to terminate without relinquishing the memory it occupies. MS-DOS reserves this area of memory by removing it from its list of free memory blocks, and loads all subsequent applications at higher memory addresses. The TSR module accesses this DOS service through the Microsoft C library function `_dos_keep`. The only complication in using this function is that it is necessary to specify the size of the program that is to be kept resident. The details of using `_dos_keep` are discussed in the next section.

The next task required to create a TSR is to establish some means for activating the program through a hot key. The basic strategy used by the TSR module is to replace the current keyboard interrupt (`09h`) handler with a Microsoft C interrupt function. Each time a key is pressed or released, the interrupt function receives control. This function first calls the former interrupt handler, and then simply tests the BIOS shift status flag to see whether the designated hot key is pressed. If the hot key is pressed, the routine performs several additional tests and possibly activates the TSR entry function.

Another important duty of the TSR code is to prevent the resident program from being activated at a time when it is not safe to interrupt the current process. The first such dangerous time is when the BIOS is engaged in disk activity. Therefore, the TSR module also installs a handler for interrupt *13h*, which is the entry point for the BIOS disk services used by MS-DOS and by some application programs. The interrupt *13h* function (in the assembly language file *TSRA.ASM*) first increments a global flag indicating that BIOS disk activity is in progress, calls the original interrupt *13h* routine, and then decrements the flag when the original routine returns. Therefore, before the keyboard interrupt handler activates the TSR entry function, it checks this flag and returns immediately if the flag is greater than zero, indicating that the disk services are currently in progress.

It is also unsafe to activate a TSR when most MS-DOS services are active. The danger of suspending an MS-DOS service is that the TSR itself may subsequently invoke an MS-DOS function, which would most likely corrupt the stack used by the original invocation of DOS. In general, the MS-DOS kernel cannot be interrupted at an arbitrary point and have its code reused by another process (since the code cannot be recursively re-entered, it is termed "nonreentrant"). MS-DOS, however, provides a partial solution to this problem. Whenever DOS is active, it sets an internal flag to a non-zero value; the address of this *INDOS* flag is returned by the undocumented, but often used, function *34h*. Therefore, in addition to checking the BIOS disk activity flag, the keyboard interrupt handler needs to check the *INDOS* flag, and return immediately if either of these flags is non-zero.

This technique to avoid interrupting MS-DOS, however, has a serious problem. While the command interpreter is waiting for input at the DOS prompt, the *INDOS* flag is set to *1*, since DOS uses one of its own services to read characters. While DOS is waiting at the command prompt, however, it is relatively safe to activate a TSR. In fact, all DOS function calls (accessed through interrupt *21h)* can be used at this time except those in the range *01h* to *0Ch*. Therefore, the TSR needs some way of knowing that although the *INDOS* flag is not *0*, DOS is simply waiting at the command line and the TSR function may be safely called. Again, MS-DOS provides a solution: while DOS waits for user input at the prompt, it continually invokes interrupt *28h* (which is used to activate background processes such as print spooling, and is known as the DOS idle interrupt). Therefore, the occurrence of interrupt *28h* is a sufficient indication (regardless of the state of the *INDOS* flag) that the TSR may be activated. Accordingly, the TSR module installs a fourth and final interrupt handler, which intercepts interrupt *28h* and provides a second door to the TSR.

In summary, there are two parallel mechanisms for activating the TSR when the hot key is pressed. First, the TSR may be activated through the keyboard interrupt handler, which must first check both the BIOS flags and the `INDOS` flag. Second, it may be activated through the interrupt `28h` handler, which needs to check only the BIOS disk activity flag.

Your TSR program can determine whether it was activated through the interrupt `28h` handler by calling the function `TsrInDos`, as described in the previous section. If this function returns `TRUE`, the `INDOS` flag is non-zero and the TSR must have been activated through the interrupt `28h` handler. In this case, your program must not call DOS functions `01h` through `0Ch`, since these functions use the same operating system stack that DOS was using when it was interrupted. The parallel mechanism for activating a TSR is illustrated in Figure 4.1.

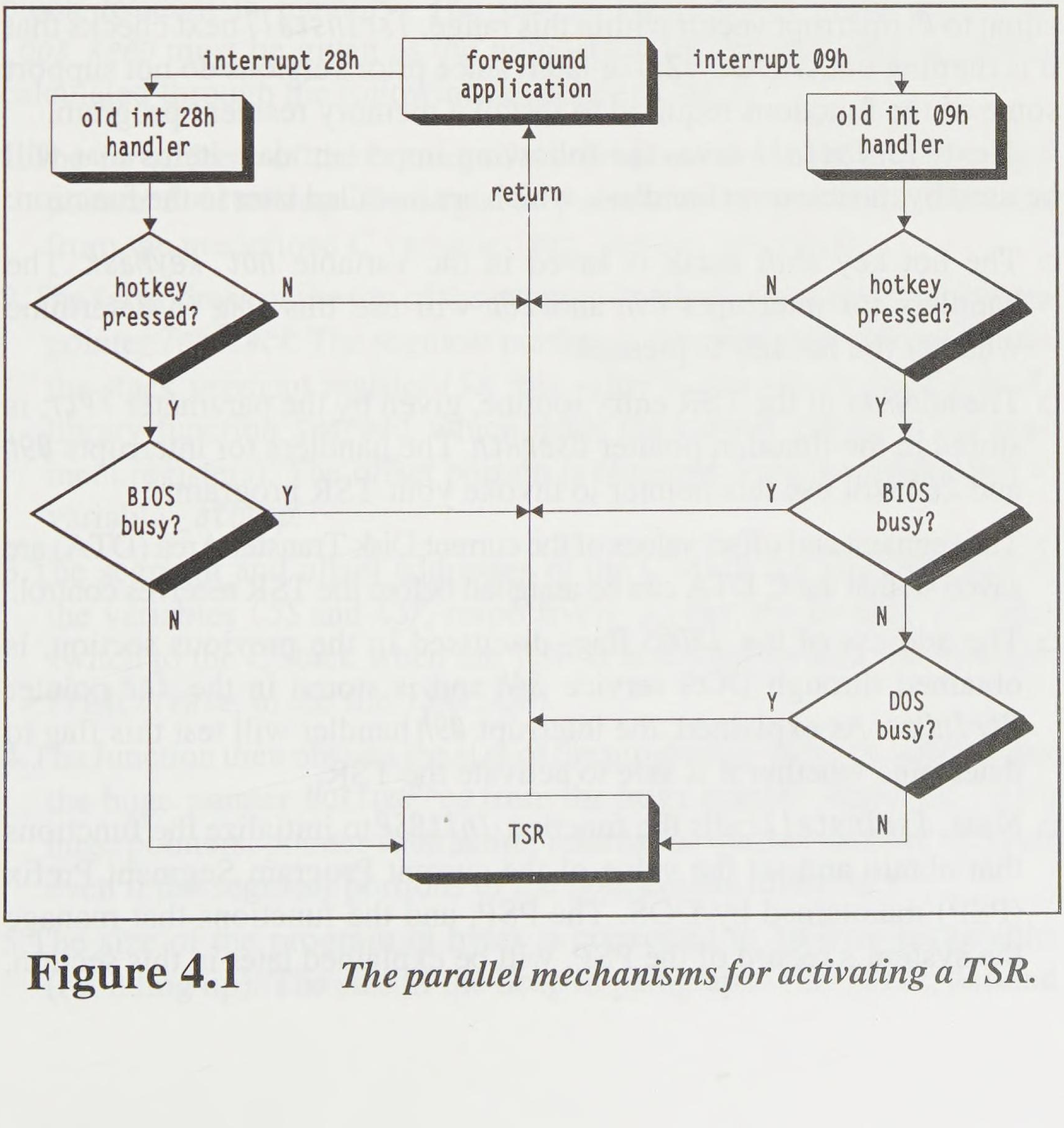

interrupt 28h
foreground application
interrupt 09h
old int 28h handler
old int 09h handler
return
hotkey pressed?
N
N
hotkey pressed?
Y
Y
BIOS busy?
Y
Y
BIOS busy?
N
N
Y
DOS busy?
TSR
N

Figure 4.1 *The parallel mechanisms for activating a TSR.*

Once it has been determined that it is safe to activate the TSR, the final major task is to save the complete machine state of the interrupted program, set up the runtime environment for Microsoft C, and call the TSR entry function that was specified at installation time. When the entry function returns, the machine state must be restored before returning to the suspended application.

Details of the Implementation

The installation function, `TsrInstall` (which is located in the C file of Listing 4.1), begins by testing to see if the TSR has already been installed in memory, by searching through the user interrupts (numbers `60h` to `67h`) for the identification number passed in the parameter `Code`. If the code is not found, indicating that the program has not already been installed, then the function proceeds to place the code value in the first unused (that is, equal to `0`) interrupt vector within this range. `TsrInstall` next checks that it is running under DOS v2.0 or later, since prior versions do not support some of the functions required to install a memory resident program.

Next, `TsrInstall` saves the following important data items that will be used by the interrupt handlers, which are installed later in the function:

- The hot key shift mask is saved in the variable `hot keyMask`. The handlers for interrupts `09h` and `28h` will use this flag to determine whether the hot key is pressed.
- The address of the TSR entry routine, given by the parameter `FPtr`, is stored in the function pointer `UserRtn`. The handlers for interrupts `09h` and `28h` will use this pointer to invoke your TSR program.
- The segment and offset values of the current Disk Transfer Area (DTA) are saved so that the C DTA can be assigned before the TSR receives control.
- The address of the `INDOS` flag, discussed in the previous section, is obtained through DOS service `34h` and is stored in the `far` pointer `PtrInDos`. As explained, the interrupt `09h` handler will test this flag to determine whether it is safe to activate the TSR.
- Next, `TsrInstall` calls the function `InitPSP` to initialize the functions that obtain and set the value of the current Program Segment Prefix (PSP) maintained by DOS. The PSP, and the functions that manage the system's record of the PSP, will be explained later in this section.

- Finally, *TsrInstall* saves the current contents of the three interrupt vectors that will subsequently be replaced.
- The values of the C stack segment and stack pointer must also be saved, so that the TSR can use the C stack rather than borrowing the stack belonging to the interrupted program. These values, however, are not stored until the necessary data are obtained in the course of calculating the size of the program, later in *TsrInstall*.

  Once the required data have been stored, *TsrInstall* installs handlers for interrupts *28h* (the DOS "idle" interrupt), *13h* (the BIOS disk services), and *09h* (the hardware keyboard interrupt).

The last task performed by *TsrInstall* is to calculate the size of the memory block occupied by the program and then to call the Microsoft C function *_dos_keep* to terminate the program while leaving the specified block resident in memory. The size of the memory block passed to *_dos_keep* must be given as the number of 16-byte paragraphs, and is calculated through the following sequence of steps:

1. The far address of the bottom of the program is assigned to the huge pointer *BottomProg*. The segment portion of this pointer is obtained from the predefined C variable *_psp*, and the offset is 0.
2. The far address of the top of the program's stack is assigned to the huge pointer *TopStack*. The segment portion of this pointer is obtained from the stack segment register (*SS*; this value is derived by calling the C library function *segread*, which reads the current values of the segment registers). The offset portion is obtained from the predefined C variable *_atopsp*.
3. The segment and offset addresses of the C stack are also assigned to the variables *CSS* and *CSP*, respectively, so that the module can later switch to the C stack when the TSR is activated (within the function *PreActivate*, in the file *TSRA.ASM*).
4. The function then obtains the size of the program in bytes by subtracting the huge pointer *BottomProg* from the huge pointer *TopStack*. (With huge pointers, pointer subtraction returns the correct number of bytes even if the segment portions of the pointers are different.)
5. The size of the program in bytes is converted to 16-byte paragraphs (rounding up). The size of the heap in paragraphs (*HEAPSIZE*, defined

in *TSR.C*) is added to this value to obtain the final number of paragraphs that should be kept in memory. The result is assigned to the variable *TSRSize*, which is passed to the C library function *_dos_keep*.

The first parameter passed to *_dos_keep* is the process return code, which is assigned to the DOS variable *ERRORLEVEL* when the program terminates.

Note that you can adjust the amount of memory reserved for the program's heap by assigning various values to the identifier *HEAPSIZE* (defined at the beginning of the C source file, *TSR.C*). Once you have developed a TSR, you can experiment to find the appropriate value for this constant. If the value is too large, the TSR will consume an undue amount of memory. If the value is too small, C memory allocation functions will overwrite memory that does not belong to the process, resulting in the DOS error message

```
Memory Allocation Error. System Halted.
```

Note that even if your program does not explicitly allocate memory from the heap, C library functions that your program calls may allocate such memory.

Figure 4.2 provides a map of the block of code and data that remains resident in memory, and illustrates the values used to calculate the size of the program.

The function *NewInt13*, located in the assembly language file of Listing 4.2, is the new handler for interrupt *13h*, which is used to access the BIOS disk services. *NewInt13* increments the value of *Int13Flag* (which is initialized to *0*), calls the original handler (the address of which was stored by *TsrInstall*), and then decrements *Int13Flag* upon return from the original handler. The flag *Int13Flag* therefore has a non-zero value only when one of the BIOS disk services is active. The program uses a counter instead of a simple true or false flag because of the small possibility that the disk services may call themselves recursively. (Note that to access *Int13Flag*, which is contained in the C data segment, the *DS* register is temporarily assigned the address of the C data segment, *DGROUP*.)

The function *NewInt09* (in the C file of Listing 4.1) is the keyboard interrupt handler. This function first calls the former interrupt handler through the function pointer *OldInt09*. It then disables the interrupts so that it will not be interrupted in the middle of testing and setting the *Busy* flag. This flag serves to prevent the TSR from being recursively re-entered while it is active. (Note that since the TSR switches to a fixed location in a local stack and uses static and external variables to store data, the code is not reentrant.) `NewInt09` next calls `GetShift` to see if the hot key is pressed. If the hot key is currently pressed, the function tests whether the BIOS disk services are active through the `Int13Flag` variable, and whether DOS is busy through the `far` pointer to the `INDOS` flag, `PtrInDos`. If all tests are passed, the function `PreActivate` is called, which performs the initial steps necessary to activate the TSR entry function. Upon return from `PreActivate`, the `Busy` flag is switched off and the interrupt handler returns.

The function `NewInt28` (also in the C file of Listing 4.1) is the interrupt `28h` handler and performs the same tasks as `NewInt09` except that it does not check whether MS-DOS is active, for reasons described in the previous section.

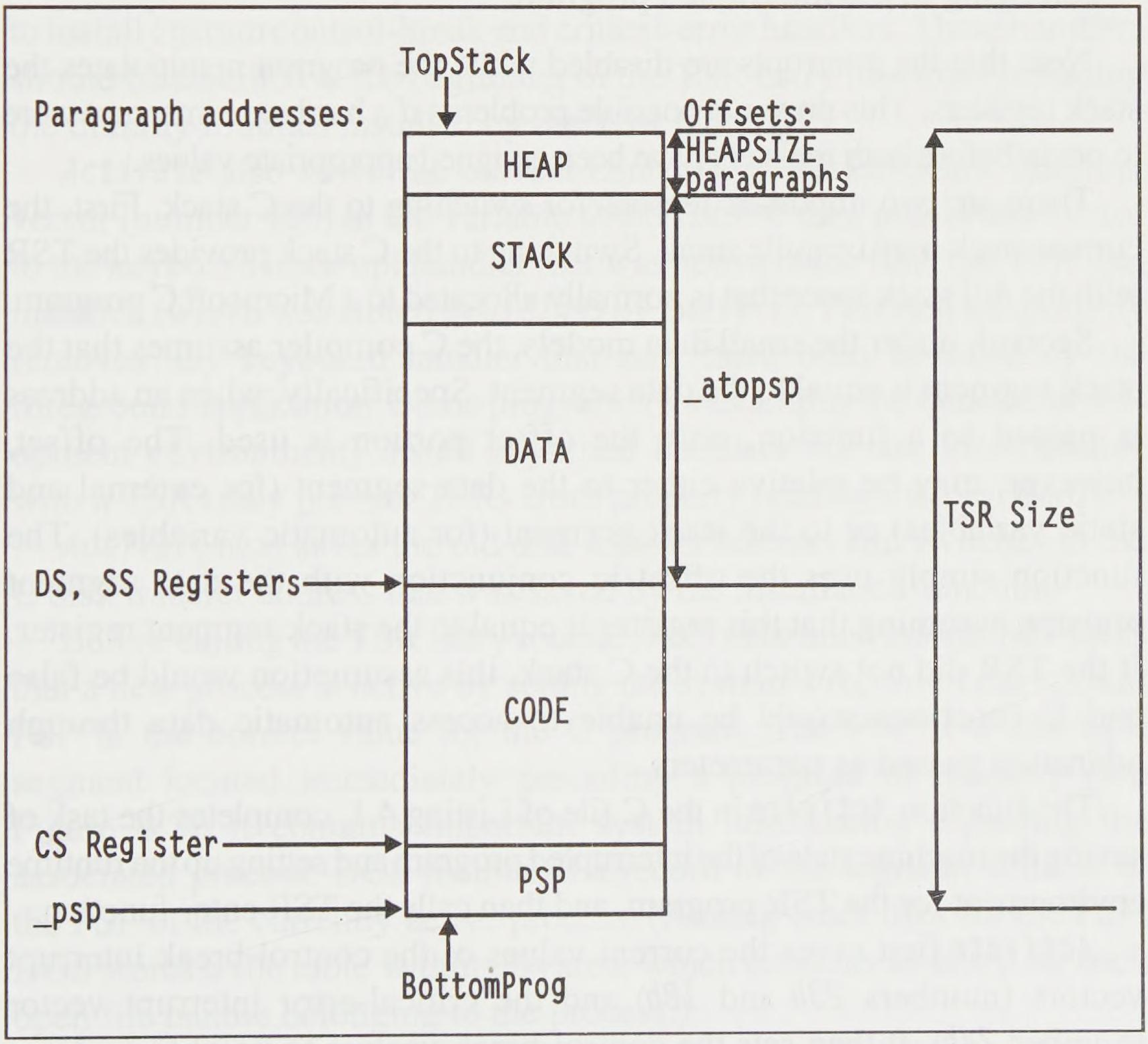

TopStack
Paragraph addresses:
Offsets:
HEAP
HEAPSIZE paragraphs
STACK
_atopsp
DATA
TSR Size
DS, SS Registers
CODE
CS Register
PSP
_psp
BottomProg

Figure 4.2 ***A memory map of the block of code and data that remain resident in memory.***

The function `PreActivate` is located in the assembly language file of Listing 4.2. It is called by `NewInt09` or `NewInt28`, when either one of these interrupt handlers has detected that the hot key is pressed and that it is safe to invoke the TSR. `PreActivate` performs the following tasks:

- It saves the existing values of the stack segment and stack pointer registers.
- It switches to the C stack by assigning `SS` and `SP` the values saved by `TsrInstall` in the variables `CSS` and `CSP` during installation.
- It calls the C function, `Activate`, which performs some additional preparations and invokes the TSR entry routine.
- Upon return from the TSR code, `PreActivate` restores the stack belonging to the interrupted program.

Note that the interrupts are disabled while the program manipulates the stack registers. This prevents possible problems if a hardware interrupt were to occur before both registers have been assigned appropriate values.

There are two important reasons for switching to the C stack. First, the current stack may be quite small. Switching to the C stack provides the TSR with the full stack space that is normally allocated to a Microsoft C program.

Second, under the small data models, the C compiler assumes that the stack segment is equal to the data segment. Specifically, when an address is passed to a function, only the offset portion is used. The offset, however, may be relative either to the data segment (for external and static variables) or to the stack segment (for automatic variables). The function simply uses the offset in conjunction with the data segment register, assuming that this register is equal to the stack segment register. If the TSR did not switch to the C stack, this assumption would be false and C functions would be unable to access automatic data through addresses passed as parameters.

The function `Activate` in the C file of Listing 4.1, completes the task of saving the machine state of the interrupted program and setting up the runtime environment for the TSR program, and then calls the TSR entry function.

`Activate` first saves the current values of the control-break interrupt vectors (numbers `23h` and `1Bh`) and the critical-error interrupt vector (number `24h`). It then sets the control-break vectors to point to routines (`NewInt23` and `NewInt1B`) that simply return without performing any action. This precaution is taken to prevent a break handler belonging to the interrupted foreground process from receiving control if the user presses `Ctrl-C` or `Ctrl-Break`. The critical-error vector is pointed to a routine (`NewInt24`) that simply assigns the value `0` to register `AX`, and then returns. This value signals DOS that the error is to be ignored. Installing this critical-error handler prevents inadvertently activating the critical-error handler belonging to the interrupted foreground program, or activating the default DOS critical- error handler. (In the event of a critical error, the default DOS handler would overwrite the screen with the familiar "Abort, Retry, Ignore, Fail?" message, and then possibly attempt to abort the TSR.)

Installing these dummy control-break and critical-error handlers prevents the certain disaster of activating an inappropriate interrupt routine. However, as mentioned previously in the article, a robust TSR may need to install custom control-break and critical-error handlers. These handlers should be installed at the beginning of the TSR entry function (replacing the dummy routines installed by the TSR module).

`Activate` also saves the current contents of the keyboard interrupt vector (number `09h`) in the variable `OldRTInt9`. It then points this vector to the keyboard interrupt handler that was active at the time the TSR was installed (which was stored in `OldInt9` by `Install`). This step temporarily removes any keyboard handler that may have been installed by the foreground application. Some programs (for example, the QuickC development environment) install keyboard handlers that are incompatible with a TSR (they prevent TSRs from properly reading the keyboard).

`Activate` next saves the old disk transfer address and switches to the C disk transfer address that was saved by the installation function.

Before calling the TSR entry routine, `Activate` must inform MS-DOS that a new process is active by setting the system's record of the current PSP to the correct value for the C program. The PSP is a 256-byte segment located immediately preceding a program in memory (see Figure 4.2); it contains important system information regarding the associated process. DOS maintains a record of the segment address of the PSP of the currently active process. (Among other uses for the PSP, DOS stores a file table within this area, which contains an entry for each open file handle belonging to the process.)

When a new program is started from the command line, or through the DOS service for executing a child process, DOS automatically updates its record of the current PSP. However, when a TSR is activated through a hot key, DOS is unaware of the change in process, and therefore you must explicitly change the system's record of the current PSP. DOS provides two undocumented functions for managing its record of the PSP. These functions are accessed through interrupt `21h`; function `51h` gets the current PSP, and function `50h` sets it. The problem with these two functions, however, is that under certain versions of DOS, they are unreliable if called when DOS is active (that is, when the `INDOS` flag is nonzero). Accordingly, the TSR module provides its own functions for getting and setting the PSP: `InitPSP`, `GetPSP`, and `SetPSP` (in the C file of Listing 4.1).

The function `InitPSP` is called by `TsrInstall` during installation of the TSR. `InitPSP` obtains a table (which has one or two entries) of the addresses within the operating system code and data segment where the values of the current PSP are stored. Once these addresses are known, `GetPSP` can obtain the current PSP by simply reading one of these addresses, and `SetPSP` can set the PSP by writing the specified value directly to each of the stored addresses.

`InitPSP` works by searching the MS-DOS code and data for bytes that match the current PSP (which is contained in the predefined C variable `_psp`). When it finds a byte that contains the value of the current PSP, it then calls function `50h` to set the PSP to some other value. If the DOS memory location changes to the new value after function `50h` is called, it must be one of the addresses where DOS stores the PSP, and therefore this address is placed in the table (`PtrPSPTable`). The routine then calls function 50h once again to restore the former PSP value. (Note that `InitPSP` may safely call DOS function `50h`, since `InitPSP` is called only from the installation routine, which is run from the DOS command line.)

`InitPSP` obtains the segment address of the beginning of the MS-DOS segment from the address of the `INDOS` pointer obtained during installation. It obtains the segment address of the end of DOS by calling another undocumented function, number `52h` (this function supplies the address of the first DOS memory control block, which immediately follows the DOS segment). Note that these two `far` addresses are converted to a common format in which the segment portion of the address is zero, so that the addresses may be numerically compared in the loop (this technique is possible since the DOS code is located entirely within the first 64Kb of memory).

Accordingly, the function `Activate` calls `GetPSP` and saves the current value of the PSP in the variable `OldPSP`.

Finally, the TSR entry function is called through the function pointer `UserRtn` that was defined in the installation procedure. Upon return from the TSR routine, `Activate` restores all values that it saved before calling the routine: the PSP, the disk transfer address, the control-break and critical-error vectors, and the keyboard vector. Activate then returns control to `PreActivate`, which switches back to the previous stack and then returns to the interrupt handler that called it (`NewInt9` or `NewInt28`).

The function `TsrInDos` simply returns the current value of the `INDOS` flag, which is obtained through the pointer `InDosPtr`.

Finally, the function `TsrRemove` causes the TSR to be removed from memory by performing the following steps:

1. It verifies that the three interrupt vectors that were set by the installation program (numbers `09h`, `13h`, and `28h`) still point to your TSR code. If one or more of these vectors now points to some new address, either the foreground application or another TSR that was subsequently installed must have reset the vector(s) to point to its own code. In this case, the foreground application or other TSR probably calls your TSR (generally, a newly installed interrupt handler should call the previous handler — a process known as chaining). If your TSR code is being called by some other process, it is obviously not safe to remove it from memory. Also, restoring the original vector values would effectively uninstall any TSR that was installed after yours. Accordingly, if one of these three vectors has changed, `TsrRemove` returns the value `CANTREMOVE` without removing the TSR.
2. `TsrRemove` restores the original handlers for interrupts `09h`, `13h`, and `28h` from the values that were saved by the installation routine (`Install`).
3. It removes the TSR signature from the user interrupt vector (by writing the value `0` to this vector).
4. It sets the flag `Unload` to `1`. When this flag is set, the function `Activate` later frees the memory blocks occupied by the TSR. `TsrRemove` does not perform this task itself, since the memory should be released immediately before exiting from the TSR. Note that a program normally occupies two memory blocks: the main block beginning at the start of the program segment prefix (`_psp`), and the block containing the program's copy of the DOS environment.

Accordingly, `Activate` must free both of these blocks (it uses the C library function `_dos_freemem`).

Listing 4.1 *C functions belonging to the TSR module.*

```
/*
     File Name:  TSR.C

     This file contains the C functions belonging to the TSR module.  The
     assembly language functions are in the file TSRA.ASM (Listing 4.2).

     Copyright (C), 1989, Michael J. Young.  All rights reserved.
*/
#include <DOS.H>
#include <STDLIB.H>
#include <MALLOC.H>

#include "TSR.H"

#pragma check_stack (off)     /* Eliminate stack checks.                    */

#define HEAPSIZE 1024         /* Memory allowed for C heap, in 16-byte      */
                              /* paragraphs.                                */

extern unsigned _atopsp;      /* Predefined C variable: offset of end of    */
                              /* C stack.                                   */

typedef void (interrupt far *VIFP) ();  /* Void Interrupt Function Pointer. */

                                   /* Variables used internally in file:    */
static int Busy = 0;               /* Flag indicates TSR is active.         */
static unsigned int CDtaOff;       /* C Disk Transfer Area offset.          */
static unsigned int CDtaSeg;       /* C Disk Transfer Area segment.         */
static int hot keyMask;             /* Hot key shift mask.                   */
static unsigned int OldDtaOff;     /* Old disk transfer address offset.     */
static unsigned int OldDtaSeg;     /* Old disk transfer address segment.    */
static void (interrupt far *OldInt1B) (void);  /* Old int 1Bh vector.       */
static void (interrupt far *OldInt23) (void);  /* Old int 23h vector.       */
static void (interrupt far *OldInt24) (void);  /* Old int 24h vector.       */
static void (interrupt far *OldInt28) (void);  /* Old int 28h vector.       */
static void (interrupt far *OldInt9) (void);   /* Old int 09h vector.       */
static void (interrupt far *OldRTInt9) (void); /* Old int 09h vector at     */
                                               /* TSR runtime.              */
static unsigned far *PtrEnvSeg;    /* Pointer to environment segment address*/
static char far *PtrInDos;         /* Pointer to DOS 'indos' flag.          */
static int Unload = 0;             /* Flag to release TSR memory.           */
static unsigned int UserInt;       /* Number of user interrupt for storing  */
                                   /* the TSR's signature.                  */
static void (*UserRtn) (void);     /* Pointer to user's start routine.      */
```

Listing 4.1 *continued*

```
                               /* Functions defined within this file:   */
static unsigned GetPSP (void);     /* Gets current PSP address from DOS.    */
int GetShift (void);               /* Gets status of shift keys.            */
static void InitPSP (void);        /* Saves DOS PSP storage locations.      */
static void interrupt far NewInt1B (void);    /* New int 1Bh handler.       */
static void interrupt far NewInt23 (void);    /* New int 23h handler.       */
                                              /* New int 24h handler:       */
static void interrupt far NewInt24 (unsigned ES, unsigned DS, unsigned DI, unsigned SI,
                                    unsigned BP, unsigned SP, unsigned BX, unsigned DX,
                                    unsigned CX, unsigned AX);
static void interrupt far NewInt28 (void);    /* New int 28h handler.       */
static void interrupt far NewInt9 (void);     /* New int 09h handler.       */
static void SetPSP (unsigned PSP); /* Sets DOS's record of PSP address.     */

                            /* Declarations shared with assembler module: */
unsigned int CSS;                  /* C stack segment.                      */
unsigned int CSP;                  /* C stack pointer.                      */
extern int Int13Flag;              /* Flag:  BIOS interrupt 13h active.     */
void (interrupt far *OldInt13) (void); /* Old int 13h vector.               */

void interrupt far NewInt13 (void);    /* New int 13h handler (in ASM mod.).*/
void far PreActivate (void);           /* Pre-activate routine (in ASM mod.)*/

void far Activate (void)
/*
     This function is called by the routine 'PreActivate' (defined in the
     assembly language module).
*/
     {
     union REGS Reg;                       /* Passes values to 'int86x'.    */
     struct SREGS SReg;                    /* Passes values to 'int86x'.    */
     unsigned OldPSP;                      /* Stores PSP address of         */
                                           /* interrupted process.          */

                                           /* Set new interrupt vectors:    */
     OldInt23 = _dos_getvect (0x23);       /* Save old break-key vector.    */
     OldInt24 = _dos_getvect (0x24);       /* Save old critical-error vector. */
     OldInt1B = _dos_getvect (0x1b);       /* Save old BIOS break vector.   */
     OldRTInt9 = _dos_getvect (0x09);      /* Save runtime int 9 handler.   */
     _dos_setvect (0x23,NewInt23);         /* Set new break-key handler.    */
     _dos_setvect (0x24,NewInt24);         /* Set new critical-error handler. */
     _dos_setvect (0x1b,NewInt1B);         /* Set new BIOS break handler.   */
     _dos_setvect (0x09,OldInt9);          /* Set old int 9 handler.        */

     Reg.h.ah = 0x2f;                      /* Save old Disk Transfer Address. */
     int86x (0x21,&Reg,&Reg,&SReg);

     OldDtaSeg = SReg.es;
     OldDtaOff = Reg.x.bx;
```

Listing 4.1 *continued*

```
     Reg.x.dx = CDtaOff;                /* Set C Disk Transfer Address.     */
     SReg.ds = CDtaSeg;
     Reg.h.ah = 0x1a;                   /* DOS set DTA service.             */
     int86x (0x21,&Reg,&Reg,&SReg);

     OldPSP = GetPSP ();                /* Get PSP address of interrupted   */
                                        /* process.                         */
     SetPSP (_psp);                     /* Set PSP for C program.           */

     (*UserRtn) ();                     /* Call the user's C function.      */

     SetPSP (OldPSP);                   /* Restore PSP for the interrupted  */
                                        /* process.                         */

     Reg.x.dx = OldDtaOff;              /* Restore Old Disk Transfer Address*/
     SReg.ds = OldDtaSeg;
     Reg.h.ah = 0x1a;
     int86x (0x21,&Reg,&Reg,&SReg);

                                        /* Restore old interrupt vectors:   */
     _dos_setvect (0x23,OldInt23);      /* Restore old break-key handler.   */
     _dos_setvect (0x24,OldInt24);      /* Restore old crit-error handler.  */
     _dos_setvect (0x1b,OldInt1B);      /* Restore old BIOS break handler.  */
     if (!Unload)
          _dos_setvect (0x09,OldRTInt9);/* Restore runtime int 9 handler.   */

     if (Unload)                        /* If TSR is being released, free   */
          {                             /* both blocks allocated to program.*/
          _dos_freemem (_psp);          /* Free the main program block.     */
          _dos_freemem (*PtrEnvSeg);    /* Free the environment block.      */
          }

     return;

     } /* end Activate */

int GetShift (void)
/*
     This function returns the BIOS shift status word.
*/
     {
     /* Far pointer to shift flag in BIOS data area:                       */
     unsigned far *PtrShiftFlag = (unsigned far *)0x00400017;

     return (*PtrShiftFlag);

     } /* end GetShift */
```

Listing 4.1 *continued*

```
static void interrupt far NewInt1B (void)
/*
     This is the new handler for interrupt 1Bh, which is invoked by the BIOS
     keyboard routine when it detects the Ctrl-break keystroke.  While the
     TSR is active, this function replaces any BIOS-level break handler that
     might have been installed by the interrupted program.
*/
     {
     return;

     } /* end NewInt1B */

static void interrupt far NewInt23 (void)
/*
     This is the new handler for interrupt 23h, which DOS activates when it
     detects a control-break key.  The function replaces the default DOS
     control-break handler, which aborts the process;  by simply returning,
     this function prevents DOS from attempting to terminate the TSR.
*/
     {
     return;

     } /* end NewInt23 */

static void interrupt far NewInt24 (unsigned ES, unsigned DS, unsigned DI, unsigned SI,
                                    unsigned BP, unsigned SP, unsigned BX, unsigned DX,
                                    unsigned CX, unsigned AX)
/*
     This is the new handler for interrupt 24h, which DOS invokes when a
     critical error occurs.  It assigns a value of 0 to AX, which informs
     DOS that the error should be ignored.  The function prevents DOS from
     attempting to abort the TSR.
*/
     {
     AX = 0;

     } /* end NewInt24 */
```

Listing 4.1 *continued*

```
static void interrupt far NewInt28 (void)
/*
     This is the new interrupt 28h, the ’DOS idle interrupt’, handler.
*/
     {
     (*OldInt28) ();          /* Chain to previous interrupt 28 handler.    */
     _disable ();             /* Disable interrupts to test and set         */
     if (Busy)                /* ’Busy’ semaphore.                          */
          return;
     Busy = 1;
     _enable ();              /* Re-enable interrupts.                      */
     if ((GetShift () & hot keyMask) != hot keyMask)    /* Test if hot key   */
          {                                            /* is pressed.       */
          Busy = 0;
          return;
          }
     if (Int13Flag)           /* Test if BIOS service is active.            */
          {
          Busy = 0;
          return;
          }

     PreActivate ();          /* Conditions are safe, therefore activate TSR*/

     Busy = 0;                /* Reset the active semaphore.                */
     return;

     } /* end NewInt28 */

static void interrupt far NewInt9 (void)
/*
     This is the new handler for interrupt 09, the hardware interrupt
     activated by the keyboard.
*/
     {
     (*OldInt9) ();      /* Chain to prior interrupt 09 handler.            */
     _disable ();        /* Disable interrupts to test and set 'Busy'       */
     if (Busy)           /* Test if TSR already active.                     */
          return;
     Busy = 1;           /* Set busy flag.                                  */
     _enable ();         /* Enable interrupts.                              */
     if ((GetShift () & hot keyMask) != hot keyMask)     /* Test if hot key   */
          {                                            /* is pressed.       */
          Busy = 0;
          return;
          }
                         /* Test if EITHER a BIOS service OR DOS is active: */
     if (Int13Flag || *PtrInDos)
          {
          Busy = 0;
          return;
          }
```

Listing 4.1 *continued*

```
     PreActivate ();          /* Conditions are safe, therefore activate TSR*/

     Busy = 0;                /* Reset the active semaphore.              */

     return;

     } /* end NewInt9 */

int TsrInDos (void)
/*
     This function returns zero if DOS is not currently active, and a
     non-zero value otherwise.
*/
     {
     return (*PtrInDos);      /* Uses the 'indos' flag maintained by DOS.  */

     } /* end TsrInDos */

int TsrInstall (void (*FPtr) (void), int hot key, unsigned long Code)
/*
     This routine terminates the C program, but leaves the code resident in
     memory.  It installs interrupt handlers so that when the shift key
     combination specified by 'hot key' is pressed, the function pointed to
     by 'FPtr' receives control, provided that conditions are safe.

     If successful, the function never returns.  If an error occurs, it
     returns one of the following error codes:

     INSTALLED      A TSR bearing the same 'Code' value has already been
                    installed.
     NOINT          No free "user" interrupts are available.
     WRONGDOS       DOS version is prior to 2.0.
     ERROR          An unidentified error has occurred.
*/
     {
     unsigned int i;                    /* Loop index.                    */
     struct SREGS SReg;                 /* Gets segment registers.        */
     union REGS Reg;                    /* Holds registers for 'int86'.   */
     char huge *TopStack;               /* Top of MSC stack.              */
     char huge *BottomProg;             /* Bottom of MSC program.         */
     unsigned TSRSize;                  /* Amount of memory to leave.     */

     UserInt = 0;
     for (i = 0x60; i <= 0x67; ++i)     /* Search "user" interrupt vectors. */
          {
          if (_dos_getvect (i) == (VIFP)Code)  /* Test for TSR signature.   */
               return (INSTALLED);
                                        /* Look for free vector:          */
          if (UserInt == 0 && _dos_getvect (i) == (VIFP)0)
               UserInt = i;
          }
     if (UserInt == 0)                  /* No free "user" interrupts.     */
          return (NOINT);
```

Listing 4.1 *continued*

```
_dos_setvect (UserInt, (VIFP)Code); /* Write TSR signature to vector. */

if (_osmajor < 2)                  /* Test that OS version is >= 2.0.  */
    return (WRONGDOS);             /* Wrong DOS version.               */
FP_SEG (PtrEnvSeg) = _psp;         /* Initialize pointer to segment    */
FP_OFF (PtrEnvSeg) = 0x2c;         /* address of TSR's environment.    */

hot keyMask = hot key;               /* Save hot key shift mask.         */

UserRtn = FPtr;     /* Save address of the routine that is activated   */
                    /* when the hot key is pressed.                    */

Reg.h.ah = 0x2f;                   /* Invoke MS-DOS function to get    */
int86x (0x21,&Reg,&Reg,&SReg);     /* current Disk Transfer Address.   */
CDtaSeg = SReg.es;
CDtaOff = Reg.x.bx;

Reg.h.ah = 0x34;                   /* Get pointer to INDOS flag        */
int86x (0x21,&Reg,&Reg,&SReg);     /* using the undocumented DOS       */
                                   /* function 34h.                    */
FP_SEG (PtrInDos) = SReg.es;
FP_OFF (PtrInDos) = Reg.x.bx;
InitPSP ();                        /* Find and save table of PSP       */
                                   /* addresses from DOS.              */
OldInt28 = _dos_getvect (0x28);    /* Save old interrupt vectors.      */
OldInt13 = _dos_getvect (0x13);
OldInt9 = _dos_getvect (0x9);
_dos_setvect (0x28,NewInt28);      /* Initialize new interrupt vectors.*/
_dos_setvect (0x13,NewInt13);
_dos_setvect (0x9,NewInt9);
                                   /* Calculate memory to keep:        */
FP_SEG (BottomProg) = _psp;        /* Set far address of bottom of TSR.*/
FP_OFF (BottomProg) = 0;

segread (&SReg);                   /* Set far address of top of stack. */
FP_SEG (TopStack) = SReg.ss;
FP_OFF (TopStack) = _atopsp;

CSS = SReg.ss;          /* Save the C stack segment and offset.        */
CSP = _atopsp;
                        /* Calculate TSR size in 16-byte paragraphs:   */
TSRSize = ((TopStack - BottomProg + 15) >> 4) + HEAPSIZE; /* Round up. */

_dos_keep (0,TSRSize);  /* Terminate and stay resident.                */

return (ERROR);         /* '_dos_keep' function should not return;     */
                        /* therefore, something has gone terribly      */
                        /* wrong.  Return the general error code.      */

} /* end TsrInstall */
```

Listing 4.1 *continued*

```
int TsrRemove     (void)
/*
     Removes the TSR from memory.  If successful, returns NOERROR (0);  if
     the TSR cannot be removed, it returns CANTREMOVE.
*/
     {
     if (OldRTInt9            != NewInt9  ||  /* First, must make sure that  */
         _dos_getvect (0x13) != NewInt13 ||  /* none of the TSR interrupts  */
         _dos_getvect (0x28) != NewInt28)    /* have been hooked by a       */
          return (CANTREMOVE);               /* subsequently loaded program.*/

     _dos_setvect (0x9,OldInt9);             /* Now restore the TSR         */
     _dos_setvect (0x13,OldInt13);           /* interrupts to their values  */
     _dos_setvect (0x28,OldInt28);           /* before TSR installation.    */

     _dos_setvect (UserInt,0);               /* Remove TSR signature from   */
                                             /* user interrupt.             */
     Unload = 1;                             /* Set flag so that 'Activate' */
                                             /* will free memory.           */
     return (NOERROR);

     } /* end TsrRemove */

/*** PSP Functions. ********************************************************/

static unsigned far *PtrPSPTable [2]; /* Table of pointers to DOS PSP values*/
static int PSPCount;                  /* Number of pointers in PtrPSPTable. */

static void InitPSP (void)
/*
     This function saves the addresses of the locations in the MS-DOS segment
     where DOS stores the current PSP value.
*/
     {
     union REGS Reg;
     struct SREGS SReg;
     unsigned char far *PtrDos;     /* Points to locations in DOS segment.   */
     unsigned char far *PtrEndDos;  /* Points to the end of DOS segment.     */
     unsigned far *PtrMCB;          /* Points to the location where the      */
                                    /* segment address of end of DOS is kept.*/

     PSPCount = 0;                  /* Initialize counter of valid entries   */
                                    /* in PtrPSPTable.                       */
```

Listing 4.1 *continued*

```
                                   /* Assign PtrDos the address of the      */
                                   /* BEGINNING of the DOS segment.         */
     FP_SEG (PtrDos) = 0;
     FP_OFF (PtrDos) = FP_SEG (PtrInDos) << 4;
                                   /* Assign PtrEndDos the address of the   */
     Reg.h.ah = 0x52;              /* END of the DOS segment.               */
     int86x (0x21,&Reg,&Reg,&SReg);
     FP_SEG (PtrMCB) = SReg.es;
     FP_OFF (PtrMCB) = Reg.x.bx - 2;
     FP_SEG (PtrEndDos) = 0;
     FP_OFF (PtrEndDos) = *PtrMCB << 4;
                                   /* Obtain addresses where DOS stores the */
                                   /* PSP of the current process.           */
     while (PtrDos < PtrEndDos && PSPCount < 2)
          {
          if (*(unsigned far *)PtrDos ==  _psp)
               {
               Reg.h.ah = 0x50;
               Reg.x.bx = _psp + 1;
               int86 (0x21,&Reg,&Reg);
               if (*(unsigned far *)PtrDos == _psp + 1)
                    PtrPSPTable [PSPCount++] = (unsigned far *)PtrDos;
               Reg.h.ah = 0x50;
               Reg.x.bx = _psp;
               int86 (0x21,&Reg,&Reg);
               }
          ++PtrDos;
          }

     } /* end InitPSP */

static unsigned GetPSP (void)
/*
     This function gets current PSP address from DOS.
*/
     {
     return *PtrPSPTable [0];

     } /* end GetPSP */

static void SetPSP (unsigned PSP)
/*
     This function sets DOS's record of PSP address.
*/
     {
     int i;

     for (i = 0; i < PSPCount; ++i)
          *PtrPSPTable [i] = PSP;

     } /* end SetPSP */

/* End of File */
```

Listing 4.2 *Assembly language functions for converting a Microsoft C program into a TSR.*

```
;    File Name:  TSRA.ASM
;
;    This file contains the assembly language functions belonging to the TSR
;    module.  The C functions are defined in the file TSR.C (Listing 4.1).
;
;    Copyright (C), 1989, Michael J. Young.  All rights reserved.

.MODEL MEDIUM

.DATA

PUBLIC _Int13Flag                            ;Flag indicating int 13h active.
       _Int13Flag  DW  0
EXTRN  _OldInt13 : DWORD                     ;Defined in RES.C.
EXTRN  _CSS      : WORD                      ;Defined in RES.C.
EXTRN  _CSP      : WORD                      ;Defined in RES.C.
       _OldSS      DW  ?                     ;Stores old stack segment.
       _OldSP      DW  ?                     ;Stores old stack pointer.

.CODE

EXTRN _Activate : FAR

PUBLIC _NewInt13
;
;    Prototype:
;         void interrupt far NewInt13 (void)
;    This is the new handler for interrupt 13h, the BIOS disk services.
;    This function maintains a flag of interrupt 13h invocations, Int13Flag.
;    Each time interrupt 13h is invoked, the flag is incremented, and each
;    time control returns from this interrupt, the flag is decremented.  A
;    zero value of Int13Flag indicates that interrupt 13h is not active.
;
_NewInt13 PROC

     push ds                                 ;Temporarily load the C data segment
     push ax                                 ;address into DS.
     mov  ax, DGROUP
     mov  ds, ax
     pop  ax

     inc  _Int13Flag                         ;Increment the interrupt 10h active flag.

     pushf                                   ;Simulate an interrupt to the old
     call _OldInt13                          ;handler.

     dec  _Int13Flag                         ;Decrement the interrupt 10h active flag.

     pop  ds                                 ;Restore the DS register.
     ret  2                                  ;Return from interrupt, preserving the flags.
_NewInt13 ENDP
```

Listing 4.2 *continued*

```
PUBLIC _PreActivate
;
;    Prototype:
;         void far PreActivate (void)
;    This is the first routine called to activate the TSR.  It performs
;    initial activation tasks and then calls the C routine Activate to
;    complete activation of the TSR.
;
_PreActivate PROC

     cli                              ;Disable interrupts.
     mov  _OldSS, ss                  ;Save the old SS and SP.
     mov  _OldSP, sp
     mov  ss, _CSS                    ;Switch to the C stack.
     mov  sp, _CSP
     sti                              ;Reenable interrupts.

     call _Activate                   ;Call C function to activate TSR.

     cli                              ;Disable interrupts.
     mov  ss, _OldSS                  ;Restore the old stack.
     mov  sp, _OldSP
     sti                              ;Reenable interrupts.

     ret                              ;Return control.

_PreActivate ENDP

END

; End of File
```

Listing 4.3 *Header file you must include in your program to call the functions in the TSR module.*

```
/*
    File Name:  TSR.H

    This file contains the function prototypes and constant definitions for
    the TSR module.  You should include this file in any program that calls
    one or more of the TSR functions.

    Copyright (C), 1989, Michael J. Young.  All rights reserved.
*/

int TsrInDos (void);
int TsrInstall (void (*FPtr) (void),int hot key, unsigned long Code);
int TsrRemove (void);

                              /* BIT MASKS FOR TSR hot key:                  */
#define RIGHTSHIFT      0x0001
#define LEFTSHIFT       0x0002
#define CONTROL         0x0004
#define ALT             0x0008
#define SCROLLLOCK      0x1000
#define NUMLOCK         0x2000
#define CAPSLOCK        0x4000
#define INSERT          0x8000

                              /* ERROR RETURN CODES:                         */
#define NOERROR     0         /* No Error.                                   */
#define ERROR       1         /* General, undefined error.                   */
#define WRONGDOS    2         /* Version of DOS prior to 2.0.                */
#define INSTALLED   3         /* TSR has already been installed in memory.  */
#define NOINT       4         /* No free user interrupts are available.      */
#define CANTREMOVE  5         /* Can't remove TSR from memory.               */

/* End of File */
```

Listing 4.4 *A simple memory resident program.*

```
/*
     File Name:  TSRDEMO.C

     This program demonstrates the following TSR functions:
          TsrInstall
          TsrInDos
          TsrRemove

     Copyright (C), 1989, Michael J. Young.  All rights reserved.
*/
#include <BIOS.H>
#include <DOS.H>
#include <MALLOC.H>
#include <MEMORY.H>
#include <PROCESS.H>
#include <STDIO.H>

#include "TSR.H"

#define ULR 6                           /* Position of centered window.      */
#define ULC 20

static void Test (void);                /* The TSR demo routine.             */

                                        /* Supporting video routines:        */
void ScrClear (int StartRow,int StartCol,int StopRow,int StopCol,int Attr);
void ScrPutBox (int UR,int LC,int LR,int RC,int Attr,int Style);
void ScrPutS (char *String,unsigned char Attr,unsigned char Row, unsigned char Col);
void ScrRestoreBlock (char *BlockAddr, int Free);
char *ScrSaveBlock (int UR, int LC, int LR, int RC);

                                        /* Constants for video colors:       */
#define FG_I        0x08                /* Lightens foreground color.        */
#define FG_R        0x04                /* Red foreground.                   */
#define FG_G        0x02                /* Green foreground.                 */

void main
     (void)
     {
     /* Install 'Test' as TSR function;  Ctrl-Left Shift is hot key.      */
     switch (TsrInstall (Test,CONTROL | LEFTSHIFT,0x11111111))
          {
          case WRONGDOS:
               printf ("Cannot install TSR;  must have DOS version 2.0 " "or higher.\n");
               exit (1);

          case INSTALLED:
               printf ("TSR already installed.\n");
               exit (1);
```

Listing 4.4 *continued*

```
          case NOINT:
               printf ("Cannot install;  no free user interrupts.\n");
               exit (1);

          case ERROR:                        /* General error case.          */
          default:
               printf ("Error installing TSR function.\n");
               exit (1);
          }

     } /* end main */

static void Test (void)
/*
     This routine is called when the hot key is pressed.  It displays a
     window and indicates whether DOS is currently active.
*/
     {
     char *ScreenHandle;
     int Key;

     ScreenHandle = ScrSaveBlock (ULR,ULC,ULR+11,ULC+39);
     ScrClear (ULR,ULC,ULR+11,ULC+39,0x0f);
     ScrPutBox (ULR,ULC,ULR+11,ULC+39,FG_I | FG_R | FG_G,2);
     ScrPutS ("T S R   D E M O",FG_I | FG_R | FG_G,ULR+2,ULC+12);
     if (TsrInDos ())
          ScrPutS ("Now in DOS",FG_I | FG_R | FG_G,ULR+5,ULC+14);
     else
          ScrPutS ("NOT in DOS",FG_I | FG_R | FG_G,ULR+5,ULC+14);
     ScrPutS ("Press R to remove TSR from memory.",FG_I | FG_R | FG_G, ULR+9,ULC+3);
     ScrPutS ("Press any other key to continue.",FG_I | FG_R | FG_G, ULR+10,ULC+4);

     Key = (_bios_keybrd (_KEYBRD_READ) & 0x00ff);

     if (Key == 'r' || Key == 'R')
          if (TsrRemove ())
               {
               ScrClear (ULR+5,ULC+1,ULR+10,ULC+38,0x0f);
               ScrPutS ("Sorry, cannot remove TSR now.",FG_I | FG_R | FG_G, ULR+5,ULC+6);
               ScrPutS ("Press any key to continue.",FG_I | FG_R | FG_G, ULR+8,ULC+7);
               _bios_keybrd (_KEYBRD_READ);
               }

     ScrRestoreBlock (ScreenHandle,1);

     } /* end Test */
```

Listing 4.4 *continued*

```
/*** Supporting video functions. ********************************************/

void ScrClear (int StartRow, int StartCol, int StopRow, int StopCol, int Attr)
/*
     This function clears the rectangular section of the screen specified
     by the four parameters.  It fills the blank area with the video display
     attribute given by 'Attr'.
*/
     {
     union REGS Reg;

     Reg.h.bh = (unsigned char) Attr;     /* BH specified video attribute.*/
     Reg.h.ch = (unsigned char) StartRow; /* CH specifies start row.      */
     Reg.h.cl = (unsigned char) StartCol; /* CL specifies start column.   */
     Reg.h.dh = (unsigned char) StopRow;  /* DH specifies stop row.       */
     Reg.h.dl = (unsigned char) StopCol;  /* DL specifies stop column.    */
     Reg.x.ax = 0x0600;       /* BIOS scroll page up function.            */
     int86 (0x10,&Reg,&Reg);  /* Invoke BIOS video services.              */

     } /* end ScrClear */

void ScrPutBox (int UR, int LC, int LR, int RC, int Attr, int Style)
/*
     This function displays a box on the screen starting at row and column
     'UR' and 'LC' and ending at row and column 'LR' and 'RC'.  It uses
     the video attribute 'Attr'.  'Style' selects the line style, and may
     be a value between 0 and 3.
*/
     {
     register int i;          /* Loop counter.                            */
     int Delta1, Delta2;      /* Bytes between lines.                     */
                              /* Store the box characters for each style. */
     static unsigned char ulc [] = {218,201,213,214};
     static unsigned char urc [] = {191,187,184,183};
     static unsigned char llc [] = {192,200,212,211};
     static unsigned char lrc [] = {217,188,190,189};
     static unsigned char hl  [] = {196,205,205,196};
     static unsigned char vl  [] = {179,186,179,186};
     unsigned far *Video;         /* Far pointer to video memory.         */
     unsigned char far *PtrVideoMode = (unsigned char far *)0x00400049;

     Delta1 = (RC - LC);           /* Characters between 2 vertical lines.*/
     Delta2 = 80 - Delta1;         /* Characters between right vertical   */
                                   /* line and left vertical line of next */
                                   /* row.                                */
```

Listing 4.4 *continued*

```
     /* Initialize far pointer to video memory, upper left corner of box: */
     FP_SEG (Video) = (*PtrVideoMode == 7) ? 0xb000 : 0xb800;
     FP_OFF (Video) = UR * 160 + LC * 2;
     *Video++ = (Attr << 8) | ulc [Style];      /* Draw upper left corner.   */
     for (i = 1; i <= RC - LC -1; ++i)          /* Draw top horizontal line. */
          *Video++ = (Attr << 8) | hl [Style];
     *Video = (Attr << 8) | urc [Style];        /* Draw upper right corner.  */
     Video += Delta2;
     for (i = 1; i <= LR - UR - 1; ++i)         /* Draw both vertical lines. */
          {
          *Video = (Attr << 8) | vl [Style]; /* Left vertical line.       */
          Video += Delta1;
          *Video = (Attr << 8) | vl [Style]; /* Right vertical line.      */
          Video += Delta2;
          }
     *Video++ = (Attr << 8) | llc [Style];      /* Draw lower left corner.   */
     for (i = 1; i <= RC - LC - 1; ++i)         /* Draw bottom horiz. line.  */
          *Video++ = (Attr << 8) | hl [Style];
     *Video = (Attr << 8) | lrc [Style];        /* Draw lower right corner.  */

     } /* end ScrPutBox */

void ScrPutS (char *String, unsigned char Attr, unsigned char Row, unsigned char Col)
/*
     This function displays null terminated 'String' with video attribute
     'Attr', beginning at the position given by 'Row' and 'Col'.
*/
     {
     register unsigned char A;           /* Fast register storage for Attr.*/
     unsigned char far *PtrVideoMode = (unsigned char far *)0x00400049;
     unsigned char far *Video;           /* Far pointer to video memory.   */

     A = Attr;                           /* Store attribute in register.   */

     /* Calculate far pointer to video memory:                            */
     FP_SEG (Video) =  (*PtrVideoMode == 7) ? 0xb000 : 0xb800;
     FP_OFF (Video) =  Row * 160 + Col * 2;
     while (*String)          /* Write characters from string until null. */
          {
          *Video++ = *String++;
          *Video++ = A;
          }

     } /* end ScrPutS */

typedef struct
     {
     int UR;
     int LR;
     int LC;
     int BytesPerRow;
     }
HEADER;
```

Listing 4.4 *continued*

```
void ScrRestoreBlock (char *BlockAddr, int Free)
/*
     This function restores to the screen a block of data saved previously
     through the function 'ScrSaveBlock'.  'BlockAddr' is the address of
     the block returned by 'ScrSaveBlock'.  If you assign the parameter
     'Free' a nonzero value, the heap memory used to store the screen data
     will be released, and you may NOT call 'ScrRestoreBlock' to restore the
     data again.  If you assign 'Free' 0, the heap memory is not released
     and you may restore the data to the screen again.
*/
     {
     register int Row;
     HEADER *PtrHeader;
     unsigned char far *PtrVideoMode = (unsigned char far *)0x00400049;
     int VideoSeg, VideoOff;
     char far *PtrBlock;

     /* Set a pointer to the header at the beginning of the block in heap:*/
     PtrHeader = (HEADER *)BlockAddr;

     /* Calculate target and source addresses:                          */
     VideoSeg = (*PtrVideoMode == 7) ? 0xb000 : 0xb800;
     VideoOff = PtrHeader->UR * 160 + PtrHeader->LC * 2;
     PtrBlock = (char far *)(BlockAddr + sizeof (HEADER));

     /* Write each row of data to the Screen:                           */
     for (Row = PtrHeader->UR; Row <= PtrHeader->LR; ++Row)
          {
          movedata                 /* Copy a row.                       */
                (FP_SEG (PtrBlock), /* Source segment.                  */
                FP_OFF (PtrBlock),  /* Source offset.                   */
                VideoSeg,           /* Target segment.                  */
                VideoOff,           /* Target offset.                   */
                PtrHeader->BytesPerRow);     /* Bytes to copy.          */

          /* Increment source and target addresses to next row:         */
          PtrBlock += PtrHeader->BytesPerRow;
          VideoOff += 160;
          }

     /* Free the block if the flag is set:                              */
     if (Free)
          free (BlockAddr);

     return;

     } /* end ScrRestoreBlock */
```

Listing 4.4 *continued*

```
char *ScrSaveBlock (int UR, int LC, int LR, int RC)
/*
     This function saves the rectangular block of screen text data specified
     through the first four parameters:

          UR   Upper row (rows are numbered beginning with 0 at the top of the screen).
          LC   Left column (columns are numbered beginning with 0 at the left of the screen).
          LR   Lower row.
          RC   Right column

     If successful, it returns the address of the block of data.  If an error
     occurs, it returns the value NULL.  The function works only in a text mode.
*/
     {
     register int Row;
     int BytesPerRow;
     size_t BlockSize;
     char *BlockAddr;
     HEADER *PtrHeader;
     unsigned char far *PtrVideoMode = (unsigned char far *)0x00400049;
     int VideoSeg, VideoOff;
     char far *PtrBlock;

     /* Calculate the size of the block and allocate memory from the heap:*/
     BytesPerRow = (RC - LC + 1)  * 2;
     BlockSize = sizeof (HEADER) + BytesPerRow * (LR - UR + 1);
     BlockAddr = (char *)malloc (BlockSize);
     if (BlockAddr == NULL)
          return (NULL);

     /* Save information on the saved block in the header at the beginning*/
     /* of the memory allocation.                                         */
     PtrHeader = (HEADER *)BlockAddr;
     PtrHeader->UR = UR;
     PtrHeader->LR = LR;
     PtrHeader->LC = LC;
     PtrHeader->BytesPerRow = BytesPerRow;

     /* Calculate initial source and target addresses:                    */
     VideoSeg = (*PtrVideoMode == 7) ? 0xb000 : 0xb800;
     VideoOff = UR * 160 + LC * 2;
     PtrBlock = (char far *)(BlockAddr + sizeof (HEADER));
```

Listing 4.4 *continued*

```
     /* Write each row of screen data to the block in the heap:            */
     for (Row = UR; Row <= LR; ++Row)
          {
          movedata                 /* Copy a row.                          */
               (VideoSeg,          /* Source segment.                      */
               VideoOff,           /* Source offset.                       */
               FP_SEG (PtrBlock),  /* Target segment.                      */
               FP_OFF (PtrBlock),  /* Target offset.                       */
               BytesPerRow);       /* Bytes to copy.                       */

          /* Increment source and target addresses for next row:           */
          PtrBlock += BytesPerRow;
          VideoOff += 160;
          }

     return (BlockAddr);      /* Return block address as a handle to the   */
                              /* saved data.                               */

     } /* end ScrSaveBlock */

/* End of File */
```

Listing 4.5 *A* MAKE *file for preparing both* TSR.C *and* TSRDEMO.C.

```
ALL : TSRDEMO.EXE

TSRDEMO.OBJ : TSRDEMO.C TSR.H
     cl /c /FPa /W2 /Zp TSRDEMO.C

TSR.OBJ : TSR.C TSR.H
     cl /c /W2 /Zp TSR.C

TSRA.OBJ : TSRA.ASM
     masm /MX TSRA.ASM;

TSRDEMO.EXE : TSRDEMO.OBJ TSR.OBJ TSRA.OBJ
     link /NOI /NOD TSRDEMO+TSR+TSRA,,NUL,SLIBCE;
```

Chapter 5

Loading TSRs into Expanded Memory

Thomas W. Olsen

One of the best features of DOS v5.0 is its ability to move TSRs, device drivers, and itself above the "640Kb barrier" and into the Upper Memory Blocks (UMBs) and High Memory Area (HMA) — leaving nearly 620Kb of free conventional memory. If you are feeling "squeezed" by DOS memory constraints, this windfall could not have arrived at a better time. Add-on products like QEMM-386 and 386MAX yield even greater savings by compressing ROMs, remapping video display memory, and squashing the BIOS data region.

Believe it or not, even more untapped DOS memory is on the horizon. This article describes how to load TSRs into expanded memory and, thus, reclaim real-mode memory for other uses. It might just eliminate the words "RAM cram" from your vocabulary. All that's required is a bit of creative — albeit unorthodox — programming. Before explaining the technique, I will briefly review DOS memory management.

DOS Memory Management

Lotus, Intel, and Microsoft formulated the original Expanded Memory Specification (LIM/EMS) in 1985 to give DOS applications more addressable memory. Bank-switched expanded memory hardware quickly faded into obscurity with the introduction of 80286 and 80386 protected-mode architectures that could directly access more than 1Mb. Microsoft soon proposed the EXtended Memory Specification (XMS) to better suit extended memory hardware. The XMS device driver (`himem.sys`) forms the cornerstone of Windows' memory management.

Nonetheless, rumors of the demise of EMS were indeed exaggerated since 80386 expanded memory managers (such as the one in Windows) simulate expanded memory hardware with extended memory; for that same reason, distinctions between the two forms of memory have become merely semantic. Furthermore, don't forget that DOS v5.0 manages the UMBs with EMS, that EMS works on virtually all PC platforms, and that thousands of DOS applications contain support for EMS, too.

The Expanded Memory Manager (EMM) device driver loads during bootup from `config.sys` and constitutes the EMS software interface to expanded memory hardware. The EMM creates a 64Kb region within the first megabyte of memory, called a "page frame" (Figure 5.1), into which applications map 16Kb "logical pages." The page frame provides a window through which real-mode applications can swap chunks of expanded memory in and out. Each block of pages has an associated handle. Applications talk to the EMM through a software interrupt (`INT 67H`) and direct all aspects of expanded memory allocation, context preservation, deallocation, frame location, and page mapping.

DOS memory management is rather crude by comparison. It hands out memory in contiguous chunks, prefacing each with a Memory Control Block (MCB) header and forming a linked list. The MCB contains a signature field ("M"=Normal Block, "Z"=End of List), the segment address of its owner (`0x0000` means the segment is free), the total number of paragraphs (16 bytes each), and the name of the owner. During memory requests, DOS sifts through the MCB chain for a free block and marks its owner field with the segment address of the requesting program. DOS deallocates a segment by resetting the MCB owner field to null.

DOS v5.0 extends the MCB chain beyond the previous version's limit of 640Kb by mapping around areas reserved for video and device ROM extensions with fake MCBs. That does not mean you will see "900,000 bytes free" after running `CHKDSK`, though. DOS v5.0 perpetuates the artificial 640Kb memory limitation for the sake of compatibility with older programs while providing separate memory management functions for the UMBs. Likewise, DOS v5.0 never loads programs above 640Kb unless told otherwise with the new `LOADHIGH` (`LH`) command, which puts them in the UMBs.

`emm386.exe` allocates UMBs with a "largest block first" methodology that sometimes leaves unused gaps in UMB memory. It takes this conservative approach because some TSRs become ill-mannered and crash the PC when confronted with marginal resources. Users often need to experiment with several different TSR-loading sequences before finding the optimal combination. QEMM386 employs so-called "stealth technology" that allows larger TSRs to squeeze into smaller UMB slots. Despite these advances, two irrefutable facts remain: there is a limited amount of UMB space and there are too many TSRs.

Now shift your attention to the EMS page frame located beside the UMBs. Its meager footprint conceals literally megabytes of additional

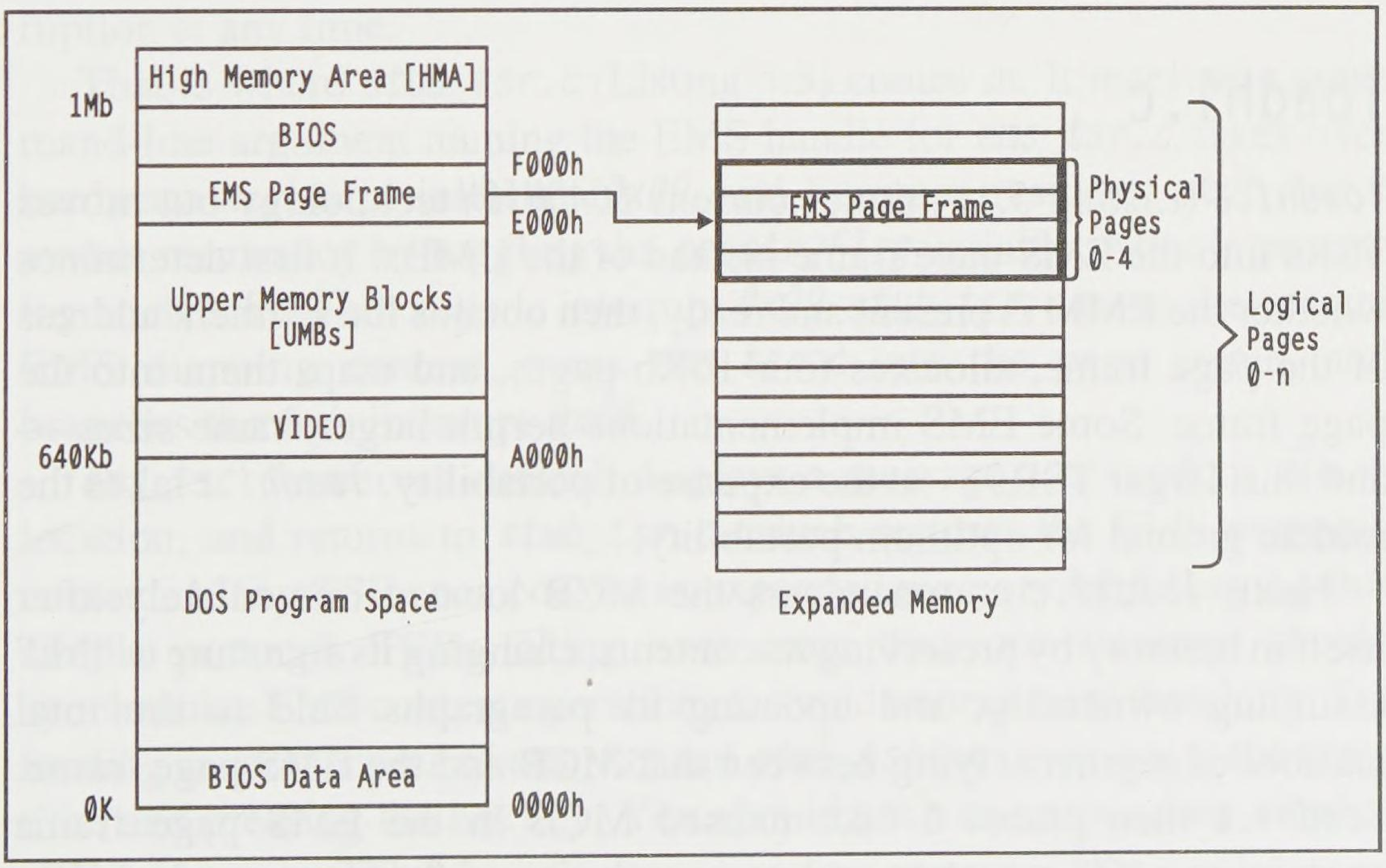

High Memory Area [HMA]
1Mb
BIOS
EMS Page Frame
F000h
E000h
Upper Memory Blocks [UMBs]
VIDEO
640Kb
A000h
DOS Program Space
BIOS Data Area
0K
0000h
EMS Page Frame
Physical Pages 0-4
Logical Pages 0-n
Expanded Memory

Figure 5.1 *Typical memory map.*

memory. Wouldn't it be great to tap those resources for DOS programs? The problem is that, unlike Windows or OS/2 programs, DOS programs do not readily disintegrate into `MOVEABLE`, `DISCARDABLE`, and `FIXED` pages. DOS reads an entire executable into fixed memory and obviates the need for intermittent page fetches. Plus, many DOS programs grab interrupt vectors and this prevents dynamic swapping. To their credit, TSR programs have simple memory requirements and virtually govern themselves after becoming resident, making them ideal candidates for expanded memory.

The next section focuses on how to place TSRs in expanded memory and ensure cooperative behavior and performance. All of the accompanying source code was compiled, linked, and tested with the Microsoft C Optimizing Compiler v6.0a. Other compilers should work with few changes, but you should pay close attention to non-ANSI standard library extensions such as `int86()` and `movedata()`. These programs require an EMS device driver. Check the documentation for your particular brand of hardware. DOS v5.0 users may include a variation on the following command in their `config.sys` file, where `nnnn` represents kilobytes of expanded memory:

```
DEVICE=EMM386.EXE RAM nnnn
```

loadhi.c

`loadhi.c` (Listing 5.1) mimics current `LOADHIGH` technology but moves TSRs into the EMS page frame instead of the UMBs. It first determines whether the EMM is present and ready, then obtains the segment address of the page frame, allocates four 16Kb pages, and maps them into the page frame. Some EMS implementations permit larger frame sizes — and thus larger TSRs — at the expense of portability. `loadhi.c` takes the middle ground for optimum portability.

Next, `loadhi.c` commandeers the MCB located immediately after itself in memory by preserving its contents, changing its signature to "M," assuming ownership, and updating its paragraphs field to the total number of segments lying between that MCB and the EMS page frame. `loadhi.c` then places a fake unused MCB in the EMS page frame containing a "Z" signature and paragraph size of `0xFFE`, or nearly 64Kb.

This strategy forces DOS to use the fake MCB in the EMS page frame for any subsequent memory operations.

`loadhi.c` loads `ems_tsr.c` (Listing 5.2) into the page frame with the `spawnvp()` function, and DOS does all of the work. DOS determines that the next block of available memory lies in the page frame, and puts `ems_tsr.c` there. `ems_tsr.c` takes over user interrupt `0x60` and becomes resident, not even aware that it resides in the page frame. Recall, however, that another application could conceivably swap the volatile TSR into expanded memory without warning. In that scenario, `ems_tsr.c` would be replaced by foreign memory pages, and interrupt `0x60` would point to nonsensical data. Anyone invoking interrupt vector `0x60` would likely crash the PC.

The fact that the TSR could get swapped out necessitates a few ground rules. `ems_tsr.c` must have a small stub program in conventional (or UMB) memory that intercepts interrupts and manages its presence in the page frame. For similar reasons, `ems_tsr.c` cannot use EMS itself without being overwritten by newly mapped pages. `ems_tsr.c` cannot service hardware interrupts or other asynchronous events (such as IPX/SPX Event Service Routines, `INT 15h` System Event Wait) that could expire while absent from memory. Stub programs can utilize EMS and asynchronous events because they are fixed in memory and subject to interruption at any time.

That is where `stub_tsr.c` (Listing 5.3) comes in. It receives a command-line argument naming the EMS handle for `ems_tsr.c`, takes over hardware keyboard interrupt `0x09`, and becomes resident. `stub_tsr.c` avoids reentrancy by toggling the `reentryFlag` variable. After intercepting an `Alt-Ctrl` hot key via interrupt `0x09`, `stub_tsr.c` saves the current EMS mapping context, maps `ems_tsr.c` into the page frame, and branches through interrupt `0x60`.

`ems_tsr.c` receives control, displays a message at a random screen location, and returns to `stub_tsr.c`, which restores the EMS mapping context. Many TSRs can coexist in expanded memory by following this simple protocol. TSRs taking over more than one interrupt should synchronize EMS context operations to avoid unnecessary thrashing. Try loading several copies of `ems_tsr` and `stub_tsr` into memory at the same time and pressing the hot key. You should see a corresponding number of screen messages.

Improvements

`loadhi.c` could probably fit `ems_tsr.c` onto a single 16Kb page, as opposed to four, by trimming the fake MCB paragraph size down to `0x3FF`. `loadhi.c` could cycle progressively through EMS page allocation, starting with one page and increasing to four pages until the TSR loads successfully. Keep in mind that EMS memory allocation occurs in chunks of 16Kb, so smaller TSRs will incur a considerable amount of overhead. Loading more TSRs into memory also means greater DOS file access requirements. Increase the `FILES` statement in the `config.sys` file accordingly.

Here's an intriguing product concept: create a loader which first places a TSR into EMS, then evaluates its interrupt usage and dynamically generates a custom stub program. That would effectively remove the requirement for hand-tailored stub programs. Special care must be taken to maintain the continuity of interrupt chaining, because some interrupts (such as `INT 21h AH=31h`) do not return to their callers. The same technology applies to device drivers as well. Imagine replacing `smartdrv.sys` or network drivers with a 1Kb stub.

Many applications improve their efficiency with overlays — groups of functions loaded and discarded upon demand from disk or memory. You could write an EMS-based TSR that provides Application Program Interface (API) services to other programs, essentially becoming a DOS Dynamic Link Library (DLL). Whenever your program requires a particular service, it interrupts the stub, which maps its DLL into the page frame and performs some relevant action. That permits multi-megabyte programs without the disk-intensive I/O of most overlay managers.

Tradeoffs

Shuttling programs between real and expanded memory incurs a slight performance penalty, measured by the inefficiency of the EMS device driver. Smart stub programs can often mitigate any performance loss by forwarding only critical events to their EMS-based partners and handling all others (and performance is, in the final analysis, a highly subjective matter). Another tradeoff is that the EMS page frame takes away 64Kb bytes from UMB memory. Unless you are loading more than 64Kb worth of EMS-based TSRs, you are actually using more memory than otherwise. Weigh these tradeoffs carefully against potential memory gains.

Conclusion

Recent advances in DOS v5.0 and other products yield additional amounts of memory, but the real-mode architecture is finite and crowding quickly. This article has shown a simple, portable, and relatively painless means of moving DOS programs into expanded memory. It doesn't rely on proprietary hardware tricks but rather on overlooked capabilities in most PCs, which gives both DOS and Windows users greater environmental flexibility.

References

Duncan, R. *Advanced MS-DOS*. Redmond, WA: Microsoft Press, 1986.

"Expanded Memory and the EMS and Extended Memory and the XMS." In *Extending DOS*, edited by Ray Duncan. Reading, MA: Addison-Wesley, 1990.

International Business Machines. *IBM Disk Operating System Technical Reference*. North Tarrytown, NY: IBM, 1987.

Microsoft Corporation. *MS-DOS Operating System Version 5.0 Users Guide and Reference*. Redmond, WA: Microsoft Corporation, 1991.

Listing 5.1 *Program to load TSR into expanded memory.*

```
/*
*************************************************************************
* Title:      LOADHI.C                                                  *
* Author:     Thomas W. Olsen                                           *
*             Copyright 1992. All Rights Reserved                       *
* Version:    1.0                                                       *
* Compiler:   Microsoft C v6.0a                                         *
*             cl /AS /Gs /W3 /Zi loadhi.c                               *
*                                                                       *
* Info:       This program loads EMS_TSR into Expanded Memory (EMS) and *
*             installs STUB_TSR in Conventional Memory.                 *
*             It is compatible with LIM EMS v3.2.                       *
*                                                                       *
*             Usage: LOADHI  STUB_TSR.EXE  EMS_TSR.EXE  [parms...]      *
*                                                                       *
*************************************************************************
*/
#include <stdlib.h>
#include <stdio.h>
#include <process.h>
#include <dos.h>
#include <string.h>
#include <fcntl.h>

#if _MSC_VER >= 600 && !defined(M_I86SM) && !defined(M_I86MM)
#error Use Small or Medium Memory Model (/AS or /AM)
#endif

#define EMS_MANAGER_STATUS      0x40
#define EMS_PAGEFRAME_SEGMENT   0x41
#define EMS_GET_NUMBER_OF_PAGES 0x42
#define EMS_ALLOCATE_PAGES      0x43
#define EMS_MAP_PAGE            0x44
#define EMS_DEALLOCATE_PAGES    0x45
#define EMS_PAGE_SIZE           16384
#define TRUE                    1
#define FALSE                   0
typedef unsigned int BOOL;
typedef unsigned int WORD;

#pragma pack(1)
typedef struct MemoryControlBlock
{
    char signature;
    WORD pspOfOwner;
    WORD paragraphs;
    char reserved[11];
} MCB;
#pragma pack()
```

Listing 5.1 *continued*

```
/***********************************************************************
* Print Error Message and Abort                                        *
***********************************************************************/
void Fail(char *message)
{
    printf("%s", message);
    exit(-1);
}

/***********************************************************************
* EMS Allocate Pages                                                   *
***********************************************************************/
BOOL EmsAllocatePages(int numpages)
{
    union REGS regs;
    regs.h.ah = EMS_ALLOCATE_PAGES;
    regs.x.bx = numpages;
    int86(0x67, &regs, &regs);
    return (regs.h.ah != 0 ? FALSE:regs.x.dx);   /* DX=handle */
}

/***********************************************************************
* EMS Deallocate Pages                                                 *
***********************************************************************/
BOOL EmsDeallocatePages(int handle)
{
    int   loop;
    union REGS regs;

    for (loop = 0, regs.h.ah = 1; loop < 5 && regs.h.ah != 0; loop++)
    {
        regs.h.ah = EMS_DEALLOCATE_PAGES;
        regs.x.dx = handle;
        int86(0x67, &regs, &regs);
        if (regs.h.ah == 0)
            return(TRUE);                           /* success */
    }
    return(FALSE);                                  /* failure */
}

/***********************************************************************
* EMS Get Page Frame Segment Address                                   *
***********************************************************************/
BOOL EmsGetPageFrame(void)
{
    union REGS regs;

    regs.h.ah = EMS_PAGEFRAME_SEGMENT;
    int86(0x67,&regs,&regs);
    return(regs.h.ah != 0 ? FALSE:regs.x.bx);   /* return segment of pageframe */
}
```

Listing 5.1 *continued*

```
/************************************************************************
* EMS Installed?                                                        *
************************************************************************/
BOOL EmsInstalled(void)
{
    BOOL  retVal = FALSE;
    int   handle;
    union REGS regs;

    if (_dos_open("EMMXXXX0", O_RDONLY, &handle) == 0)
    {
        regs.h.ah = 0x44;                   /* dos function 44h */
        regs.h.al = 0x00;
        regs.x.bx = handle;                 /* file handle      */
        int86(0x21,&regs,&regs);            /* issue interrupt  */
        _dos_close(handle);
                                            /* Error or Not Device Driver? */
        if ((regs.x.cflag & 0x01) == 0 && (regs.x.dx & 0x80))
        {
            regs.h.ah = EMS_MANAGER_STATUS;
            int86(0x67, &regs, &regs);      /* Is Driver Ready for Action? */
            retVal = (regs.h.ah == 0) ? TRUE:FALSE;
        }
    }
    return(retVal);
}

/************************************************************************
* EMS Map Logical Page to Physical Page                                 *
************************************************************************/
BOOL EmsMapPage(int handle, int physical_page, int logical_page)
{
    union REGS regs;
    regs.h.ah = EMS_MAP_PAGE;
    regs.h.al = (char) physical_page;
    regs.x.bx = logical_page;
    regs.x.dx = handle;
    int86(0x67, &regs, &regs);
    return(regs.h.ah == 0 ? TRUE:FALSE);
}

/************************************************************************
* Peek Memory Into Buffer                                               *
************************************************************************/
void Peek(int segment, int offset, unsigned length, void far *buffer)
{
    movedata(segment, offset, FP_SEG(buffer), FP_OFF(buffer), length);
}
```

Listing 5.1 *continued*

```
/*********************************************************************
* Poke Buffer Into Memory                                            *
*********************************************************************/
void Poke(int segment, int offset, unsigned length, void far *buffer)
{
   movedata(FP_SEG(buffer), FP_OFF(buffer), segment, offset, length);
}

/*********************************************************************
* Main Program                                                       *
*********************************************************************/
int main(int argc, char *argv[])
{
   MCB     saveMcb, mcb;
   WORD    emsPageFrame, emsHandle, emsPages;
   unsigned loop, segment;
   char    parameter[20];

   /*****************************************************************
   * Correct Parms? Is EMS Installed & Ready?                         *
   *****************************************************************/
   if (argc <= 2)
      Fail("\nUSAGE: LOADHI stubTSR emsTSR [parms ...]\n");

   if (EmsInstalled() == FALSE)
      Fail("\nEMS Not Available.\n");

   /*****************************************************************
   * Allocate and MAP 4 EMS Pages (4 * 16K = 64K)                     *
   *****************************************************************/
   emsPageFrame = EmsGetPageFrame();       /* Locate Page Frame */
   emsPages     = 4;

   if ((emsHandle = EmsAllocatePages(emsPages)) == FALSE)  /* enough EMS? */
      Fail("Could Not Allocate EMS Pages.\n");

   for (loop = 0; loop < emsPages; loop++)  /* map 4 allocated pages */
      EmsMapPage(emsHandle, loop, loop);

   /*****************************************************************
   * Save a Copy of the Memory Control Block Located After Us         *
   *****************************************************************/
   segment = _psp - 1;
   Peek(segment, 0, sizeof(MCB), &mcb);
   segment += (mcb.paragraphs + 1);
   Peek(segment, 0, sizeof(MCB), &saveMcb);
```

Listing 5.1 *continued*

```
    /*****************************************************************
    * Make the Next Memory Control Block Unusable                    *
    *****************************************************************/
    Peek(segment, 0, sizeof(MCB), &mcb);
    mcb.signature  = 'M';
    mcb.pspOfOwner = _psp;
    mcb.paragraphs = emsPageFrame - segment - 1;       /* Next MCB is EMS Page Frame */
    Poke(segment, 0, sizeof(MCB), &mcb);

    /*****************************************************************
    * Put a Fake MCB in the EMS Page Frame.                          *
    *****************************************************************/
    mcb.signature  = 'Z';
    mcb.pspOfOwner = 0;
    mcb.paragraphs = 0xFFE;
    Poke(emsPageFrame, 0, sizeof(MCB), &mcb);

    /*****************************************************************
    * Load TSR in EMS Page Frame                                     *
    *****************************************************************/
    spawnvp(P_WAIT, argv[2], &argv[3]);    /* EMS_TSR [parms ...] */

    /*****************************************************************
    * Restore Conventional Memory to its Original State              *
    *****************************************************************/
    Poke(segment, 0, sizeof(MCB), &saveMcb);

    /*****************************************************************
    * Load Stub TSR in Conventional Memory                           *
    *****************************************************************/
    sprintf(parameter, "%d", emsHandle);
    execlp(argv[1], argv[1], parameter, NULL);  /* DOS_TSR emshandle */
    return(0);
}

/* End of File */
```

Listing 5.2 *Body of expanded memory TSR.*

```
/*
**************************************************************************
* Title:      EMS_TSR.C                                                  *
* Author:     Thomas W. Olsen                                            *
*             Copyright 1992. All Rights Reserved                        *
* Version:    1.0                                                        *
* Compiler:   Microsoft C v6.0a                                          *
*             cl /AS /Gs /W3 /Zi EMS_TSR.C                               *
* Info:       This program is loaded by LOADTSR.C into the EMS page      *
*             frame. When invoked via INT 60H, it displays a message     *
*             at a random screen location.                               *
*                                                                        *
**************************************************************************
*/
#include <dos.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>

void Prntln(int row, int column, char *message, int color)
{
    int far *videoPtr = (int far *) (0xB8000000L + (row * 160 + column * 2));
    int loop;

    for (loop = 0; message[loop] != '\0'; loop++, videoPtr++)
        *videoPtr = ((color << 8) | message[loop]); /* Direct Video Write */
    return;
}

void (interrupt _far Int60Handler)(void)
{
    int row = (rand() & 15), column = (rand() & 63);  /* Random row & column */

    Prntln(row, column, "Hello from EMS!", 78);
    return;
}

int main(int argc, char *argv[])                /* Here's the TSR format: */
{                                               /*         CODE        */
    extern unsigned end;                        /*      STATIC DATA    */
    unsigned _far *upperLimit;                  /*         STACK       */
    unsigned paragraphs, actual, stackSize;     /*         HEAP        */

    upperLimit = &end;                          /* top of static data */
    stackSize  = ((stackavail() + 2048) / 2048) * 2048;  /* compute stack size */
     paragraphs = (FP_SEG(upperLimit) - _psp) + (FP_OFF(upperLimit) + stackSize) / 16 + 1;
    _dos_setvect(0x60, Int60Handler);           /* set INT 60h vector to
        _dos_setblock(paragraphs, _psp, &actual);  /* relinquish unused heap */
    _dos_keep(0, paragraphs);                   /* terminate & stay resident */
    return(-1);                                 /* error must have occurred */
}

/* End of File */
```

Listing 5.3 *Conventional memory stub for* EMS_TSR

```
/*
**************************************************************************
* Title:      STUB_TSR.C                                                 *
* Author:     Thomas W. Olsen                                            *
*             Copyright 1992. All Rights Reserved                        *
* Version:    1.0                                                        *
* Compiler:   Microsoft C v6.0a                                          *
*             cl /AS /Gs /W3 /Zi STUB_TSR.C                              *
*                                                                        *
* Info:       This program is loaded by LOADHI.EXE into conventional    *
*             memory and serves as a conduit to the EMS_TSR program in   *
*             expanded memory. STUB_TSR is responsible for saving the    *
*             EMS context, mapping EMS_TSR into the page frame,          *
*             branching to its entrypoint, and restoring the EMS         *
*             context on the way out. LOADHI.EXE passes the emsHandle    *
*             for EMS_TSR as a commandline parameter to STUB_TSR.        *
*                                                                        *
*             Usage: STUB_TSR emsHandle        (ie. STUB_TSR 1 )         *
*                                                                        *
**************************************************************************
*/
#include <dos.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>

typedef     unsigned int BOOL;
#define     TRUE    1
#define     FALSE   0

void        (interrupt far *OldKeyboardIntHandler)(void);
unsigned    EmsHandle;

void (interrupt _far KeyboardIntHandler)(void)
{
    static char far *keyboardShiftFlags = (char far *) 0x00400017L;
    static union    REGS regs;
    static int      loop;
    static BOOL     reentryFlag = FALSE;

    (*OldKeyboardIntHandler)();                  /* Forward to Previous Int Handler */

    if (reentryFlag == TRUE ||                   /* Are we becoming Reentrant? */
        (*keyboardShiftFlags & 0x0C) != 0x0C)    /* <Alt-Ctrl> Pressed? */
        return;

    reentryFlag = TRUE;

    regs.h.ah = 0x47;                    /* LIM EMS: Save Mapping Context */
    regs.x.dx = EmsHandle;
    int86(0x67, &regs, &regs);
```

Listing 5.3 *continued*

```
    for (loop = 0; loop < 4; loop++)
    {
        regs.h.ah = 0x44;               /* LIM EMS: Map Page */
        regs.h.al = (char) loop;        /* Set Physical Page No. */
        regs.x.bx = loop;               /* Set Logical Page No. */
        regs.x.dx = EmsHandle;
        int86(0x67, &regs, &regs);
    }

    int86(0x60, &regs, &regs);          /* Call the TSR in the EMS page frame */

    regs.h.ah = 0x48;                   /* LIM EMS: Restore Mapping Context */
    regs.x.dx = EmsHandle;
    int86(0x67, &regs, &regs);

    reentryFlag = FALSE;
}

int main(int argc, char *argv[])                /* Here's the TSR format: */
{                                               /*         CODE        */
    extern unsigned end;                        /*      STATIC DATA    */
    unsigned _far *upperLimit;                  /*         STACK       */
    unsigned paragraphs, actual, stackSize;     /*         HEAP        */

    if (argc < 2)
        return(-1);

    EmsHandle  = atoi(argv[1]);
    upperLimit = &end;                          /* top of static data */
    stackSize  = ((stackavail() + 2048) / 2048) * 2048; /* compute stack size */
    paragraphs = (FP_SEG(upperLimit) - _psp) + (FP_OFF(upperLimit) + stackSize) / 16 + 1;
    OldKeyboardIntHandler = _dos_getvect(0x09); /* get old INT 2Fh vector */
    _dos_setvect(0x09, KeyboardIntHandler);     /* set INT 2Fh vector to our handler */
        _dos_setblock(paragraphs, _psp, &actual);   /* relinquish unused heap */
    _dos_keep(0, paragraphs);                   /* terminate & stay resident */
    return(-1);                                 /* error must have occurred */
}

/* End of File */
```

Chapter 6

Making Windows and DOS Programs Talk

Thomas W. Olsen

With countless copies of Windows in circulation, software developers are frantically scrambling to move their DOS applications to Windows. Many developers need techniques that allow the Windows and DOS worlds to "talk" to one another. Unfortunately, programs cloistered in a DOS box cannot ordinarily converse with Windows siblings through DDE, messages, and global memory. Simple file-based communication gets the job done but incurs a significant performance penalty for time-conscious applications.

Not all communication techniques are available or appropriate for every Windows operating mode, as shown in Table 6.1. All the Windows modes (real, standard, and enhanced) support cooperative multitasking for Windows applications, but real and standard modes can only run one DOS application at a time, and no other programs run while that single DOS application is running. That means that if your DOS application sends data to a Windows application, you have to manually switch back to Windows before the Windows application can receive the data and act on it. Only in enhanced mode is Windows capable of preempting a DOS application and giving time slices to both Windows applications and other DOS programs. Therefore, although you can use real and standard modes for simple communications, enhanced mode is a requirement for fast, two-way, asynchronous communication between DOS and Windows applications.

What if Windows and DOS programs could directly exchange text and graphical Clipboard data, share the same DLL functions, execute one another, send messages, or establish DDE conversations? Life would be a little easier. This chapter focuses on three basic communication methods, the first and second involving a shared memory buffer and the third describing how to call a Windows program directly from DOS. I compiled the programs in this chapter with Microsoft C v6.0a, the Microsoft Windows Software Development Kit (SDK), and the Microsoft Windows Device Development Kit (DDK).

Using the Clipboard

The Windows Clipboard makes it easy to "cut and paste" text and graphics with a few mouse clicks. Contrary to popular belief, the Clipboard is not merely the Windows `clipbrd.exe` program, but a shared memory area that acts as a temporary repository for data. The Clipboard works like an ordinary shared data file. You must open it (which temporarily locks out other applications), check its contents, read or write data, and close it (making it available again to other applications) upon completion. The Clipboard supports seven data formats (`TEXT`, `BITMAP`, `METAFILE`, `SYLK`, `DIF`, `TIFF`, and `OEM TEXT`), and you can even register your own proprietary formats.

	Real	**Standard**	**Enhanced**
Clipboard	—	—	✓
TSR	✓	✓	✓
Virtual device	—	—	✓

Table 6.1 ***Cross reference to various techniques.***

The Windows SDK documents the Windows API functions for manipulating the Clipboard, but Microsoft also offers an obscure Clipboard interface (`WINOLDAP`) for DOS programs running in a Windows enhanced-mode DOS box. The SDK does not document the `WINOLDAP` interface, but it is fairly simple; DOS programs need only initialize a few registers and execute an `INT 2Fh` to access the Windows clipboard. `clip.c` (Listing 6.1) illustrates a few typical operations. It opens the Clipboard, checks for text data, and, depending on the result, displays or updates the contents. Note that `WINOLDAP` represents the maximum Clipboard size with a long integer since large bitmaps easily exceed 64Kb.

Keep a few points in mind if you use the `WINOLDAP` interface. Verify the existence of `WINOLDAP` with `INT 2Fh` before invoking any Clipboard function calls from DOS. This is also a useful way to determine whether you are running in Windows enhanced mode — `WINOLDAP` does not exist in real and standard modes. Do not leave the Clipboard open for extended periods since that prevents other applications from using it. Empty the Clipboard before writing new data to it or Windows will append the new data to the existing data. Also, before retrieving something from the Clipboard, make sure that the target buffer is big enough to hold the resulting data. IBM Common User Access (CUA) guidelines discourage programs from updating the Clipboard without an explicit user action like a CUT, COPY, or PASTE mouse click; otherwise, you could interfere with someone using the clipboard interactively in another application.

The Clipboard is a cut and paste buffer, nothing more. It is too volatile for bidirectional program communication because anyone can update the Clipboard at any moment. You could treat the Clipboard like a dirty communications buffer, filtering out "line noise" from foreign applications, but that violates CUA standards and good taste. Used properly, the Clipboard is quick, convenient, and very powerful. DOS editors, word processors, hypertext, and desktop publishing programs could implement the `WINOLDAP` Clipboard interface to provide some Windows compatibility with minimal source code changes.

Using a TSR

Unlike the Clipboard, which requires enhanced mode and provides minimal data protection, the TSR approach to communications works in every Windows operating mode and makes shared data more discrete; only conversant applications will know about the shared buffer. If you load a device driver or TSR before you run Windows, it becomes part of the System VM and Windows will map a single copy of that code to each subsequent VM (that is, DOS box) that you create. All the VMs "see" the TSR in their address spaces and consider it their own, even though it is physically mapped to the System VM. Windows and DOS programs can communicate with the TSR via a software interrupt to obtain a shared memory buffer address (within the TSR) and manipulate its contents freely. Changes to the shared buffer affect all VMs.

`tsr.c` (Listing 6.2) loads before Windows, hooks interrupt vector `2Fh`, becomes resident, waits for calls matching multiplex handle `0xBEEF`, and forwards all others to the original `INT 2Fh` interrupt handler. `INT 2Fh` is a safe choice because DOS inevitably takes it over during bootup. You could use `INT 60h` or some other user-defined interrupt, but callers have no assurances that it, in fact, points to `tsr.c` or anyone at all. `tsr.c` determines its resident memory requirements by summing the size of the PSP and the segments containing code, static data, the stack (rounded up from `stackavail()` to the nearest 2Kb), and the near heap. Microsoft always places DOS program segments in the same order except in custom memory models.

On the Windows side, `tsrwin.c` (Listing 6.4) creates a window, ascertains whether Windows is in real or protected mode with `GetWinFlags()`, and performs `INT 2Fh` with the `0xBEEF` multiplex handle. `IntHandler()` receives and modifies the full complement of registers as parameters. It returns the real-mode address of the shared memory buffer in `DX:AX` and its length in `CX`. Note that `IntHandler()` does not return the address in segment registers because those are destroyed during processor mode switches. The buffer is declared as `static char far *` for simplicity. The module definition file for the Windows TSR interface is in Listing 6.3 (`tsrwin.def`).

DOS and real-mode Windows programs can use the address returned by the interrupt handler to modify the buffer. However, protected-mode Windows programs do not understand real-mode addresses. `tsrwin.c` therefore allocates and initializes a (LDT) selector with undocumented Windows kernel functions `AllocSelector()`, `SetSelectorBase()`, and `SetSelectorLimit()`. `SetSelectorBase()` takes advantage of the fact that the linear address is equivalent to the real-mode physical address. `tsrwin.c` passes the resulting selector to the `MAKELONG` macro and builds a `char far *` pointer. The program can then read and write directly to real-mode memory from protected mode.

There are several notable drawbacks to the TSR approach. Loading the TSR before Windows means a loss of valuable conventional memory within all VMs. The Clipboard occupies conventional memory but only temporarily before disappearing. On the bright side, DOS v5.0 can load TSRs into Upper Memory Blocks above 640Kb. That leaves more free memory for applications. Loading the TSR after you run Windows will not work because the TSR would only appear in one particular VM, not all VMs. Programs may not interrupt or call code segments in separate VMs. Furthermore, `tsrwin.c` depends on undocumented Windows kernel functions for real-mode address conversion. Microsoft makes no guarantees about the veracity and longevity of undocumented function calls.

`tsr.c` reveals a few arbitrary design decisions. The shared buffer does not exceed 64Kb presently, but there is no reason, aside from the loss of conventional memory, that you cannot make the buffer larger. The `0xBEEF` multiplex handle is purely whimsical. Formulating stupid words from hexadecimal characters such as `0xBABE`, `0xFACE`, and `0xDEAD`, may lighten the mood of the next person who inherits your code. Unlike the Clipboard, `tsr.c` does not serialize access to the shared buffer. You need some kind of mechanism, whether "open" and "close" functions in `IntHandler()` or a "busy bit" buffer prefix, to keep programs from blasting each other's data. Those decisions are perhaps best left to applications using the TSR approach.

Using Virtual Device Drivers

Every PC contains a plethora of devices: the keyboard, disk drives, monitor, printer, mouse, and timer, to name a few. Windows enhanced mode furnishes Virtual Device Drivers (VxDs) as a means of "virtualizing" the access of multitasking VMs to those devices. For example, when an application writes data to a file but the request cannot be fulfilled immediately, VxDs allow the application to resume while rescheduling the disk write. Another example: the Virtual Keyboard Driver reads keystrokes from the hardware, inserts them into the active VM's keyboard buffer, detects a VM task switch, and alters its focus to the new VM. Think of a VxD as a "traffic cop" for multitasking device accesses.

As shown in Figure 6.1, the Windows DPMI server and VxDs run outside the VM architecture at the highest possible privilege level ("ring zero"). This permits them to alter the contents of VMs, twiddle the hardware, allocate memory, switch VMs, schedule events, manipulate interrupts, and do many other things. Windows and DOS applications

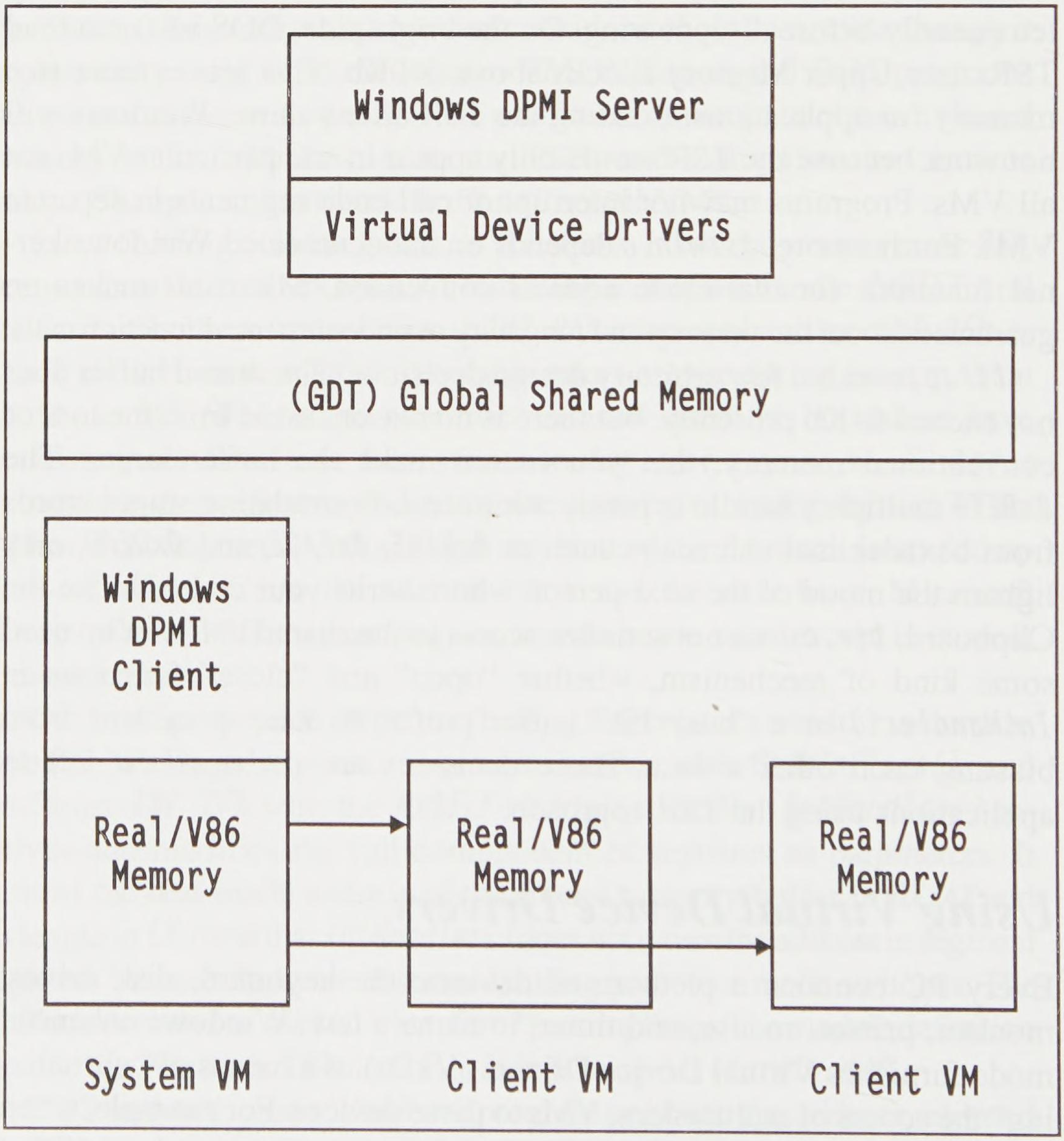

Windows DPMI Server
Virtual Device Drivers
(GDT) Global Shared Memory
Windows DPMI Client
Real/V86 Memory
Real/V86 Memory
Real/V86 Memory
System VM
Client VM
Client VM

Figure 6.1 *Windows DPMI client-server architecture.*

execute at the lowest privilege level ("ring three") and generate exception interrupts (manifested as UAEs) for violating their privilege level. VxDs are 32-bit assembly language programs produced with the Microsoft MASM5 32-Bit Assembler, `LINK386`, and `ADDHDR` tools in the Microsoft Device Development Kit (DDK). `ADDHDR` prepends a special header to the VxD that identifies it as a Virtual Device Driver. Windows loads VxDs during startup. All you have to do is specify the "`DEVICE=pipe.386`" statement in the "`[386Enh]`" section of the `system.ini` file.

Developing VxDs is a complicated business. The Windows DDK contains precious little documentation on the mechanics of VxDs. Sure, it tells you how to use specific system calls but does not sufficiently describe their overall context in the Windows architecture. One of the best sources of information is the Microsoft Operating Systems Forum (`GO MSOPSYS`) on Compuserve. Its members often remind me that building VxDs is truly a trial and error approach, even for seasoned veterans. Be prepared to reboot your machine relentlessly until the bloody things work. You can debug VxDs by hooking up a remote terminal PC via a null modem cable, loading `wdeb386.exe` on the target PC, and interspersing lots of `INT 1h` and `INT 3h` breakpoints in the VxD code.

Once you get the hang of things, you will find that VxDs have some splendid capabilities. The rich macro library provides unmatched access to system resources, the Windows DPMI server, and other VxDs. System macros are generally preceded by the term `VMMCall` and followed by parameters strung between angle brackets (such as `VMMCall_Allocate_GDT_Selector <edx, eax, 0>`). Since these terms expand from macros into low-level interrupts, the resulting assembly language looks nothing like the original system call.

The PIPE *VxD*

Shared memory is a desirable device for communications between DOS and Windows, since it is the fastest way to transfer data. Operating systems (like Windows NT) generally use shared memory to implement interprocess communications such as events, semaphores, and message passing. The TSR approach provides shared memory, but uses up precious conventional memory in all VMs. A more difficult problem with the TSR approach is that both sides must poll the buffer.

`pipe.asm` (Listing 6.6) is a VxD that attacks both these problems. With this VxD, your DOS program can give a Windows program access to any real-mode location in its VM, not just to TSR buffer space in the System VM. Your DOS program can also use this VxD to send a window message to a Windows program. This section explains how `pipe.asm` provides a simple, efficient, and elegant communications channel between Windows and DOS.

`pipe.asm` resembles a normal DOS device driver in many ways. The `Declare_Virtual_Device` macro expands into a device header containing the name, version, ID, control procedure, and API procedures for real and protected mode. OEMs are supposed to register their device IDs with Microsoft before distributing VxDs to the world populace. The device ID is a unique number that identifies the vendor and type of device. If you are serious about this stuff, apply for a valid device ID. The control procedure is like an interrupt handler that monitors the state of VMs while keeping a watchful eye on some physical device. Rather than keep track of each VM, `PIPE_Device_Init` stores "instance" information in each VM's control block, as shown in the template `VM_Control_Block_Data`.

VxDs can provide services to both DOS and Windows applications. Windows advertises the `PIPE_API` service entry point address through the `INT 2Fh` interrupt, just like `WINOLDAP` does. To determine the entry point of a particular VxD, simply pass its unique device ID to `INT 2Fh` with the appropriate function number. `pipe.asm` offers a rather different service, though. It enables a DOS program to call a Windows program. Here is the basic idea: `pipedos.c` (Listing 6.7) calls a service in `pipe.asm`, which in turn calls Windows program `pipewin.c` (Listing 6.9). Both `pipedos.c` and `pipewin.c` use `pipefunc.c` (Listing 6.10) to determine the VxD entry point address, set up a few service-related registers, call the entry point, emerge at `PIPE_API`, and perform the corresponding `PIPE_` service. Client registers are retrieved from the EBP-based stack frame.

How does `pipe.asm` know which Windows procedure to call? `pipewin.c` (Listing 6.9) first creates a window and registers `CallbackProc()` with `PIPE_RegisterCallback()`, passing its window handle and callback address as parameters. `PIPE_RegisterCallback()` saves the window handle in `WinProc_hWnd`, and the `CallbackProc()` address in `Winproc_CallbackCS` and `WinProc_CallbackIP`. `pipe.asm` does not currently allow more than one Windows callback procedure, but you could adapt it for that purpose. `pipewin.c` goes into hibernation, waiting for a DOS program to invoke its `CallbackProc()`. `pipedos.c` (Listing 6.7) allocates a protected-mode selector with `PIPE_AllocateSelector()` and points it to a real-mode buffer. `pipedos.c` must pass a protected-mode pointer to `pipewin.c` because the latter cannot determine which VM contains the buffer; GDT selectors are shared by all VMs.

Next, `pipedos.c` calls `PIPE_Call_Windows_Procedure()` with parameters `WM_USER` and the protected-mode selector. The constant `WM_USER` informs the window procedure that the DOS application is passing a buffer pointer. `PIPE_Call_Windows_Procedure()` verifies that a Windows callback was registered and that the System VM is active — branching to procedures in an absent VM crashes the system. `PIPE_Call_Windows_Procedure()` schedules `PIPE_Event_Procedure()` to pop up during the next task switch to the System VM. `PIPE_Event_Procedure()` then pushes `WinProc_hWnd`, `WM_USER`, and the protected-mode selector onto the stack and calls `CallbackProc()`.

Sounds like the end of the story, right? Wrong. The System VM is active but in a nonetheless unpredictable state. It seems that the Windows kernel functions are not reentrant, and the only function your callback procedure can safely call is `PostMessage()`. That is where the window handle enters the picture. It allows `PostMessage()` to send `WM_USER` and the protected-mode pointer to `WindowProc()` after the System VM settles into a stable state. `WindowProc()` receives the message and displays the buffer contents in a window. Meanwhile, `pipedos.c` resumes execution without waiting for `WindowProc()` to complete. If it expects a return value from `WindowProc()`, `pipedos.c` should establish and poll a "busy bit" inside the buffer. `WindowProc()` should set its return value in the buffer and toggle the busy bit off upon completion.

The module definition file for the VxD is in Listing 6.5 (`pipe.def`) and the module definition file for the Windows program is in Listing 6.8 (`pipewin.def`). The code segment definitions specify, respectively, `NONDISCARDABLE` and `FIXED` to prevent Windows from discarding the `PIPE_` services and `CallbackProc()` from memory. Discardable code segments have a tendency to move around in memory during paging operations. `PIPE_RegisterCallback()` assumes that `CallbackProc()` occupies a fixed code location. Likewise, `PIPE_UnregisterCallback()` needs to know when `pipewin.c` terminates, to avoid calling an invalid code location.

Summary

This example may help you realize the immense potential of Virtual Device Drivers. Such power and flexibility often blurs the lines between VxDs and the Windows DPMI server. You can extend DDE conversations to DOS applications, share common DLLs, and transfer text and graphics between cooperating programs. By building on `pipe.asm`, you can quickly tack a Windows user interface onto an existing DOS application and postpone the cost of porting it to Windows. [Editor's note: see Chapter 29 for more on building a Windows interface with a virtual device driver.]

References

DPMI Specification Committee. *DOS Protected Mode Interface Specification (DPMI) Version 1.0.* Santa Clara, California: DPMI Specification Committee: 1991.

Duncan, R. *Advanced MS-DOS.* Redmond, Washington: Microsoft Press, 1986.

Intel Corporation. *386 DX Microprocessor Programmer's Reference Manual.* Santa Clara, California, 1986.

International Business Machines Corp. *SAA Common User Access: Advanced Interface Design Guide.* Armonk, New York, 1989.

Microsoft Corporation. *Microsoft Windows DDK Virtual Device Adaptation Guide.* Redmond, Washington: Microsoft, 1990.

Norton, P. and Wilton, R. *The New Peter Norton Programmer's Guide to the IBM PC and PS/2.* Redmond, Washington: Microsoft Press, 1988.

Petzold, C. *Programming Windows: The Microsoft Guide for Writing Applications for Windows 3.* Redmond, Washington: Microsoft Press, 1990.

Listing 6.1 `clip.c`

```
#define  MAIN
#include <dos.h>
#include <stdlib.h>
#include <malloc.h>
/*
*********************************************************
Author:     Thomas W. Olsen
Version:    3.0
Compiler:   Microsoft C 6.0
            cl /AL /Zi clip.c
*********************************************************
*/
#define GET_WINOLDAP_VERSION        0x1700
#define OPEN_CLIPBOARD              0x1701
#define EMPTY_CLIPBOARD             0x1702
#define SET_CLIPBOARD_DATA          0x1703
#define GET_CLIPBOARD_DATA_SIZE     0x1704
#define GET_CLIPBOARD_DATA          0x1705
#define CLOSE_CLIPBOARD             0x1708
#define COMPACT_CLIPBOARD           0x1709

#define TEXT_FORMAT                 1
#define BITMAP_FORMAT               2
#define METAFILE_FORMAT             3
#define SYLK_FORMAT                 4
#define DIF_FORMAT                  5
#define TIFF_FORMAT                 6
#define OEM_TEXT                    7

int ErrorExit(char *message)
{
    printf(message);
    exit(-1);
}

int main(int argc, char *argv[])
{
    long clipboardSize;
    char far *clipboardBuffer;
    union REGS regs;
    struct SREGS sregs;

    /****************************************************/

    regs.x.ax = GET_WINOLDAP_VERSION;
    int86x(0x2F, &regs, &regs, &sregs);

    if (regs.x.ax == GET_WINOLDAP_VERSION)
        ErrorExit("Windows Not Loaded or Clipboard Not Supported.\n");
```

Listing 6.1 *continued*

```
    /*****************************************************/

    regs.x.ax = OPEN_CLIPBOARD;
    int86x(0x2F, &regs, &regs, &sregs);

    if (regs.x.ax == 0)
        ErrorExit("Clipboard in Use.\n");

    /*****************************************************/

    regs.x.ax = GET_CLIPBOARD_DATA_SIZE;            /* AX = Function No. */
    regs.x.dx = TEXT_FORMAT;                        /* DX = Data Format */
    int86x(0x2F, &regs, &regs, &sregs);

    if ((clipboardSize = ((long) regs.x.dx << 4L) + regs.x.ax) == 0L)
    {
        if ((clipboardBuffer = malloc(1024)))
        {
            strcpy(clipboardBuffer, "This is a test of the Clipboard!!!");

            regs.x.ax = SET_CLIPBOARD_DATA;         /* AX = Function No. */
            regs.x.dx = TEXT_FORMAT;                /* DX = Data Format */
            sregs.es  = FP_SEG(clipboardBuffer);    /* ES:BX -> Clip Buffer */
            regs.x.bx = FP_OFF(clipboardBuffer);
            regs.x.si = 0;                          /* SI:CX = (long) Size */
            regs.x.cx = strlen(clipboardBuffer);
            int86x(0x2F, &regs, &regs, &sregs);
        }
    }
    else
    {
        clipboardBuffer = malloc((int) clipboardSize);

        regs.x.ax = GET_CLIPBOARD_DATA;             /* AX = Function No. */
        regs.x.dx = TEXT_FORMAT;                    /* DX = Data Format */
        sregs.es  = FP_SEG(clipboardBuffer);        /* ES:BX -> Clip Buffer */
        regs.x.bx = FP_OFF(clipboardBuffer);
        int86x(0x2F, &regs, &regs, &sregs);

        if (regs.x.ax != 0)                         /* Successful? */
            printf("Clipboard Contains [%s]\n", clipboardBuffer);
    }

    regs.x.ax = CLOSE_CLIPBOARD;                    /* AX = Function No. */
    int86x(0x2F, &regs, &regs, &sregs);
    free(clipboardBuffer);                          /* Clean Up */
}

/* End of File */
```

Listing 6.2 `tsr.c`

```
#include <dos.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>
/*
*******************************************************************
Title:      TSR.C - DOS Component of TSR Interface
Author:     Thomas W. Olsen
Version:    3.0
Compiler:   Microsoft C 6.0
            cl /AS /Zi tsr.c
*******************************************************************
*/
void   (interrupt far *OldIntHandler)();

void (interrupt _far IntHandler)(int es, int ds, int di, int si, int bp,
                                 int sp, int bx, int dx, int cx, int ax,
                                 int ip, int cs, int flags)
{
    static char _far *buffer = {"These are the contents of the TSR buffer!"};

    if (ax != 0xBEEF)                        /* If Multiplex Handle doesn't */
        _chain_intr(OldIntHandler);          /*  match ... forward the interrupt */

    dx = FP_SEG(buffer);                     /* DX:AX = Buffer address */
    ax = FP_OFF(buffer);
    cx = _fstrlen(buffer);                   /* CX    = Max. buffer length */
}

int main(int argc, char *argv[])                 /* Here's the TSR format: */
{                                                /*         CODE        */
    extern unsigned end;                         /*      STATIC DATA    */
    unsigned _far *upperLimit;                   /*         STACK       */
    unsigned paragraphs, actual, stackSize;      /*         HEAP        */

    upperLimit = &end;                           /* top of static data */
    stackSize  = ((stackavail() + 2048) / 2048) * 2048; /* compute stack size */
    paragraphs = (FP_SEG(upperLimit) - _psp) + (FP_OFF(upperLimit) + stackSize) / 16 + 1;
    OldIntHandler = _dos_getvect(0x2F);   /* get old INT 2Fh vector */
    _dos_setvect(0x2F, IntHandler);       /* set INT 2Fh vector to our handler */
        _dos_setblock(paragraphs, _psp, &actual);   /* relinquish unused heap */
    _dos_keep(0, paragraphs);                    /* terminate & stay resident */
    return(-1);                                  /* error must have occurred */
}

/* End of File */
```

Listing 6.3 tsrwin.def

```
NAME           TSRWIN

DESCRIPTION    'TSR Data Transfer Example'

EXETYPE        WINDOWS

STUB           'WINSTUB.EXE'

CODE           PRELOAD MOVEABLE
DATA           PRELOAD MOVEABLE MULTIPLE

HEAPSIZE       2048
STACKSIZE      4096

EXPORTS
               WindowProc    @1
```

Listing 6.4 tsrwin.c

```
#include <dos.h>
#include "windows.h"
/*
*******************************************************************
Title:      TSRWIN.C - Windows Component of TSR Interface
Author:     Thomas W. Olsen
Version:    3.0
Compiler:   Microsoft C 6.0
            cl /c /AL /Gsw /W3 /Oas /Zpe /Zi tsrwin.c
            link /CO /NOD tsrwin,,, libw llibcew kernel, tsrwin.def
            rc tsrwin.exe
*******************************************************************
*/
long FAR    PASCAL WindowProc(HWND hWnd, unsigned message, WORD wParam, LONG lParam);
int         PASCAL SetSelectorBase(int, DWORD);
int         PASCAL SetSelectorLimit(int, DWORD);
HANDLE      PASCAL AllocateSelector(unsigned segment, unsigned offset, DWORD length);

int PASCAL WinMain(HANDLE hInst, HANDLE hPrevInst, LPSTR lpCmdLine, int numCmdShow )
{
    MSG       msg;
    HWND      hWnd;
    HDC       hContext;
    HANDLE    selector = 0;
    LPSTR     lpBuffer = NULL;
    WNDCLASS  wc;
    union     REGS regs;

    /************************** Setup Window ************************/

    wc.style         = NULL;
    wc.lpfnWndProc   = WindowProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInst;
    wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = "TsrWinClass";

    if (!RegisterClass(&wc))
        return(FALSE);

    hWnd = CreateWindow( "TsrWinClass", "Windows & TSR Example",
                          WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, NULL, NULL, hInst, NULL );
    if (!hWnd)
        return (FALSE);

    ShowWindow( hWnd, numCmdShow );
    hContext = GetDC( hWnd );                    /* Need Display Context */
```

Listing 6.4 *continued*

```
    /*************************** Call TSR **************************/

    regs.x.ax = 0xBEEF;                         /* AX = Multiplex Handle */
    regs.x.bx = 0;
    int86(0x2F, &regs, &regs);                  /* Call DOS App via INT 2FH */

    /************************ Display Buffer ***********************/

    if (regs.x.ax != 0xBEEF)                    /* Where's the Beef? */
    {
        if (GetWinFlags() & WF_PMODE)           /* Running in Protected Mode? */
        {
            selector = AllocateSelector(regs.x.dx, regs.x.ax, (DWORD) regs.x.cx);
            lpBuffer = (LPSTR) MAKELONG(0, selector);
        }
        else                                    /* No ... Must Be Real Mode */
            lpBuffer = (LPSTR) MAKELONG(regs.x.ax, regs.x.dx);

        if (lpBuffer)
            TextOut( hContext, 0, 0, lpBuffer, regs.x.cx ); /* Print Buffer */

        if (selector)                           /* If Selector Allocated, */
            FreeSelector(selector);             /* Free it ... Demo's Over */
    }
    else
    {
        lpBuffer = "TSR Not Loaded.";
        TextOut( hContext, 0, 0, lpBuffer, lstrlen(lpBuffer) );
    }

    ReleaseDC( hWnd, hContext );                /* Free Display Context */
    UpdateWindow( hWnd );

    while (GetMessage(&msg, NULL, NULL, NULL))  /* Typical Message Loop */
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return (msg.wParam);
}

long FAR PASCAL WindowProc(HWND hWnd, unsigned message, WORD wParam, LONG lParam)
{
    switch (message)
    {
        case WM_DESTROY:
            PostQuitMessage( NULL );
            break;
        default:
            return (DefWindowProc(hWnd, message, wParam, lParam));
    }
}
```

Listing 6.4 *continued*

```
HANDLE PASCAL AllocateSelector(unsigned segment, unsigned offset, DWORD length)
{
    DWORD physicalAddress, linearAddress;       /* This function allocs & maps */
    HANDLE selector;                            /* a protected mode selector */
    struct SREGS sregs;                         /* for a real mode address */

    segread(&sregs);                            /* Copy DS selector */

    if ((selector = AllocSelector(sregs.ds)))   /* Successful Allocate? */
    {                                           /* Phys = Lin In 1st Megabyte */
        linearAddress = physicalAddress = ((DWORD) segment << 4L) + offset;
        SetSelectorBase(selector, linearAddress);  /* Set New Base Address */
        SetSelectorLimit(selector, length);     /* Set New Segment Range */
    }
    return(selector);
}

/* End of File */
```

Listing 6.5 pipe.def

```
LIBRARY     PIPE

DESCRIPTION 'Pipe Virtual Device Driver v3.0'

EXETYPE     DEV386

SEGMENTS
              _LTEXT PRELOAD NONDISCARDABLE
              _LDATA PRELOAD NONDISCARDABLE
              _TEXT  CLASS 'PCODE' NONDISCARDABLE
              _DATA  CLASS 'PCODE' NONDISCARDABLE

EXPORTS
            PIPE_DDB
```

Listing 6.6 pipe.asm

```
        ;*************************************************************
        ; Title         PIPE.ASM - Pipe Virtual Device Driver
        ; Version:      3.00
        ; Author:       Thomas W. Olsen
        ; Assembler:    Microsoft Macro Assembler 5.10B
        ; Linker:       Microsoft Linear-Executable Linker 1.00.058
        ;               masm5 -p -w2 -Mx -Zd pipe;
        ;               link386 pipe.obj,pipe.386 /NOI /NOD /NOP,,,pipe.def;
        ;               addhdr pipe.386
        ;*************************************************************

        .386p
        INCLUDE VMM.Inc

        PIPE_DEVICE_ID         EQU 2020H
        PIPE_MAJOR_VERSION     EQU 03
        PIPE_MINOR_VERSION     EQU 00
        PIPE_VERSION_NO        EQU (PIPE_MAJOR_VERSION * 256 + PIPE_MINOR_VERSION)
        GET_VERSION            EQU 0
        REGISTER_CALLBACK      EQU 1
        UNREGISTER_CALLBACK    EQU 2
        ALLOCATE_GDT_SELECTOR  EQU 3
        FREE_GDT_SELECTOR      EQU 4
        CALL_WINDOWS_PROCEDURE EQU 5

        ;*************************************************************
        ;                       Device Header
        ;*************************************************************

Declare_Virtual_Device  PIPE, PIPE_MAJOR_VERSION, PIPE_MINOR_VERSION,      \
                        PIPE_CONTROL, PIPE_DEVICE_ID, Undefined_Init_Order, \
                        PIPE_API, PIPE_API

        ;*************************************************************
        ;                    Virtual Machine Data
        ;*************************************************************
        VM_Control_Block_Data struc
                msgNo           dw ?            ;Client_AX
                wParam          dw ?            ;Client_BX
                lParamLo        dw ?            ;Client_CX
                lParamHi        dw ?            ;Client_DX
        VM_Control_Block_Data ends
```

Listing 6.6 *continued*

```
        ;*************************************************************
        ;                       Local Data
        ;*************************************************************
VxD_DATA_SEG

        VM_Control_Block_Offset dd ?
        WinProc_Installed       dw 0
        WinProc_hWnd            dw ?
        WinProc_Callback_CS     dw ?
        WinProc_Callback_IP     dw ?

        EXPORT_TABLE            label dword
                                dd offset32 PIPE_Get_Version
                                dd offset32 PIPE_Register_CallBack
                                dd offset32 PIPE_Unregister_CallBack
                                dd offset32 PIPE_Allocate_GDT_Selector
                                dd offset32 PIPE_Free_GDT_Selector
                                dd offset32 PIPE_Call_Windows_Procedure
        NO_OF_EXPORTS           EQU ($ - EXPORT_TABLE) / 4

VxD_DATA_ENDS

        ;*************************************************************
        ;                      Control Procedures
        ;*************************************************************
VxD_LOCKED_CODE_SEG
        BeginProc PIPE_Device_Init
                VMMcall _Allocate_Device_CB_Area, <<SIZE VM_Control_Block_Data>, 0>
                mov     [VM_Control_Block_Offset], eax
                clc
                ret
        EndProc PIPE_Device_Init

        BeginProc PIPE_Control
                Control_Dispatch Device_Init,   PIPE_Device_Init
                clc
                ret
        EndProc PIPE_Control
VxD_LOCKED_CODE_ENDS

        ;*************************************************************
        ;                           APIs
        ;*************************************************************
VxD_CODE_SEG
```

Listing 6.6 *continued*

```
;*************************************************************
;On Entry: AX = Function Number to Execute
;*************************************************************
BeginProc PIPE_API,PUBLIC
        and     [ebp].Client_Flags, CF_Mask
        movzx   eax, [ebp].Client_AX
        cmp     ax, NO_OF_EXPORTS
        jae     short @@A99
        jmp     EXPORT_TABLE[eax * 4]
@@A99:
        ret
EndProc PIPE_API

;*************************************************************
;On Exit:    AX = Version Number
;*************************************************************
BeginProc PIPE_Get_Version,PUBLIC
        mov     [ebp].Client_AX, PIPE_VERSION_NO
        and     [ebp].Client_Flags, NOT CF_Mask
        ret
EndProc PIPE_Get_Version

;*************************************************************
;On Entry:  CX:DX = Windows Proc CS:IP EntryPoint
;           BX    = Window Handle
;*************************************************************
BeginProc PIPE_Register_CallBack,PUBLIC
        mov     WinProc_Installed, 1
        mov     cx,[ebp].Client_CX
        mov     WinProc_Callback_CS, cx
        mov     dx,[ebp].Client_DX
        mov     WinProc_Callback_IP, dx
        mov     bx,[ebp].Client_BX
        mov     WinProc_hWnd, bx
        and     [ebp].Client_Flags, NOT CF_Mask
        ret
EndProc PIPE_Register_CallBack

;*************************************************************
;On Entry:  None
;*************************************************************
BeginProc PIPE_Unregister_CallBack,PUBLIC
        mov     WinProc_Installed, 0
        and     [ebp].Client_Flags, NOT CF_Mask
        ret
EndProc PIPE_Unregister_CallBack
```

Listing 6.6 *continued*

```
;**************************************************************
;On Entry:  BX = Real Mode Segment Register
;On Exit:   AX = Protected Mode Selector
;**************************************************************
BeginProc PIPE_Allocate_GDT_Selector,PUBLIC
        Client_Ptr_Flat eax, bx   ;Map a Selector to DOS partition
                                  ; using [ebp].Client_DS
        VMMCall _BuildDescriptorDWORDs, <eax, 0FFFFh, RW_Data_Type,
        VMMCall _Allocate_GDT_Selector, <edx, eax, 0>
        mov     [ebp].Client_AX, ax
        and     [ebp].Client_Flags, NOT CF_Mask
        ret
EndProc PIPE_Allocate_GDT_Selector

;**************************************************************
;On Entry:  BX    = Protected Mode Selector
;**************************************************************
BeginProc PIPE_Free_GDT_Selector,PUBLIC
        mov     bx, [ebp].Client_BX
        movzx   ebx, bx
        VMMCall _Free_GDT_Selector, <ebx, 0>     ;EBX = GDT Selector
        and     [ebp].Client_Flags, NOT CF_Mask
        ret
EndProc PIPE_Free_GDT_Selector

;**************************************************************
;On Entry:  BX    = (WORD) msgNo
;           SI    = (WORD) wParam
;           CX:DX = (LONG) lParam
;**************************************************************
BeginProc PIPE_Call_Windows_Procedure,PUBLIC
        cmp     WinProc_Installed, 0            ;If Not Installed, Abort
        je      short @@C2
        mov     esi, ebx                        ;EBX = VM Handle
        add     esi, [VM_Control_Block_Offset]  ; point to that VM's area
        mov     ax, [ebp].Client_BX
        mov     [esi].msgNo, ax
        mov     ax, [ebp].Client_SI
        mov     [esi].wParam, ax
        mov     ax, [ebp].Client_CX
        mov     [esi].lParamHi, ax
        mov     ax, [ebp].Client_DX
        mov     [esi].lParamLo, ax
```

Listing 6.6 *continued*

```
                VMMcall Test_Sys_VM_Handle              ;Are We in the System VM?
                jnz     short @@C1
                VMMcall Get_Crit_Section_Status         ;Is Something Critical Happening?
                jc      short @@C1
                mov     edx, ebx                        ;EDX = Current VM Handle
                mov     esi, offset32 PIPE_EventProcedure
                VMMcall Call_When_VM_Ints_Enabled       ;Call Event Procedure Now
                jmp     short @@C2
        @@C1:
                mov     edx, ebx                        ;EDX = Current VM Handle
                VMMcall Get_Sys_VM_Handle               ;EBX = System VM Handle
                mov     eax, High_Pri_Device_Boost      ;EAX = Priority Boost
                mov     ecx, PEF_Wait_For_STI or PEF_Wait_Not_Crit ;ECX = Flags
                mov     esi, offset32 PIPE_EventProcedure   ;ESI = Offset of Event Procedure
                VMMcall Call_Priority_VM_Event          ;Post Event in Task Queue
        @@C2:
                ret
        EndProc PIPE_Call_Windows_Procedure

        ;*************************************************************
        ;On Entry:  EDX = Current VM Handle
        ;*************************************************************
        BeginProc PIPE_EventProcedure, PUBLIC
                Push_Client_State
                mov     esi, edx                        ;EDX = Current VM Handle
                add     esi, VM_Control_Block_Offset
                VMMCall Begin_Nest_Exec                 ;Setup Call Frame

                mov     ax, WinProc_hWnd                ;Push Parms onto Stack
                VMMcall Simulate_Push                   ;Using PASCAL Convention (4,3,2,1)
                mov     ax, [esi].msgNo
                VMMcall Simulate_Push
                mov     ax, [esi].wParam
                VMMcall Simulate_Push
                mov     ax, [esi].lParamHi
                VMMcall Simulate_Push
                mov     ax, [esi].lParamLo
                VMMcall Simulate_Push
                mov     cx, WinProc_Callback_CS
                mov     dx, WinProc_Callback_IP
                movzx   edx, dx
                VMMCall Simulate_Far_Call               ;Push Address Onto Stack

                VMMCall Resume_Exec                     ;Call Windows Procedure
                VMMcall End_Nest_Exec
                Pop_Client_State
                ret
        EndProc  PIPE_EventProcedure

VxD_CODE_ENDS

END
; End of File
```

Listing 6.7 pipedos.c

```
#define MAIN
#include <dos.h>
#include <string.h>
#include "windows.h"    /* This isn't required ... It was convenient */
/*
*****************************************************************
Title:      PIPEDOS.C - DOS Component of Pipe Interface
Author:     Thomas W. Olsen
Version:    3.0
Compiler:   Microsoft C 6.0
            cl /AL /Gs /Zi /c PIPEDOS.C
            cl /AL /Gs /Zi /c PIPEFUNC.C
            link /CO PIPEDOS PIPEFUNC;
*****************************************************************
*/

BOOL PASCAL PipeSetup(void);
WORD PASCAL PipeAllocateSelector(WORD segment);
void PASCAL PipeFreeSelector(WORD selector);
BOOL PASCAL PipeCallWindowsProc(WORD message, WORD wParam, LONG lParam);

struct _AppInfo
{
    BOOL BusyFlag;
    char Buffer[128];
}
AppInfo;

int main(int argc, char *argv[])
{
    WORD  selector;
    LPSTR protectedModePtr;
    union REGS regs;

    if (PipeSetup() == TRUE)  /* Find Version & Entrypoint of Pipe */
    {
        AppInfo.BusyFlag = TRUE;
        strcpy(AppInfo.Buffer, "These are the contents of the real mode buffer!");

        /* Windows cannot use real mode addresses ... so we need to allocate
           a protected mode GDT selector */

        selector         = PipeAllocateSelector(HIWORD(&AppInfo));
        protectedModePtr = (LPSTR) MAKELONG(LOWORD(&AppInfo), selector);

        PipeCallWindowsProc(WM_USER,                    /* This function calls the */
                            _fstrlen(AppInfo.Buffer), /* Pipe ... which calls the */
                            (LONG) protectedModePtr);   /* Windows Proc */
```

Listing 6.7 *continued*

```
    /* You cannot do a PipeFreeSelector() until you're certain that the
       Windows proc has used it ... otherwise, you'll get a UAE. Have
       the Windows proc set a flag in the buffer you're passing to it
       which signifies completion. */

    while (AppInfo.BusyFlag == TRUE)
    {
        kbhit();                              /* Gotta do Something... */
        regs.x.ax = 0x1680;                   /* WINDOWS API */
        int86(0x2F, &regs, &regs); /* Release Time Slice */
    }

    PipeFreeSelector(selector);
  }
  else
    printf("Pipe Not Found\n");
}

/* End of File */
```

Listing 6.8 pipewin.def

```
NAME          PIPEWIN

DESCRIPTION   'Windows Component of Pipe Interface v3.0'

EXETYPE       WINDOWS

STUB          'WINSTUB.EXE'

CODE          PRELOAD FIXED
DATA          PRELOAD MOVEABLE MULTIPLE

HEAPSIZE      2048
STACKSIZE     4096

EXPORTS
              WindowProc    @1
```

Listing 6.9 pipewin.c

```
#include <dos.h>
#include "windows.h"
/*
*******************************************************************
Title:      PIPEWIN.C - Windows Component of Pipe Interface
Author:     Thomas W. Olsen
Version:    3.0
Compiler:   Microsoft C 6.0
            cl /AL /Gsw /W3 /Oas /Zpe /Zi /c pipewin.c
            cl /AL /Gsw /W3 /Oas /Zpe /Zi /c pipefunc.c
            link /CO /NOD pipewin pipefunc ,,, libw llibcew kernel, pipewin.def
            rc pipewin.exe
*******************************************************************
*/
void FAR    PASCAL CallbackProc(HWND hWnd, unsigned message, WORD wParam, LONG lParam);
long FAR    PASCAL WindowProc(HWND hWnd, unsigned message, WORD wParam, LONG lParam);
BOOL        PASCAL PipeSetup(void);
BOOL        PASCAL PipeRegisterCallback(HWND hWnd, FARPROC lpProcPtr);
BOOL        PASCAL PipeUnregisterCallback(void);


int PASCAL WinMain(HANDLE hInst, HANDLE hPrevInst, LPSTR lpCmdLine, int numCmdShow )
{
    MSG       msg;
    HWND      hWnd;
    HDC       hContext;
    HANDLE    selector = 0;
    LPSTR     lpBuffer = NULL;
    WNDCLASS  wc;

    /************************ Setup Window ***********************/

    wc.style         = NULL;
    wc.lpfnWndProc   = WindowProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInst;
    wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = "PipeClass";
```

Listing 6.9 *continued*

```
    if (!RegisterClass(&wc))
        return(FALSE);

    hWnd = CreateWindow( "PipeClass", "Windows / DOS Pipe Example",
                          WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, NULL, NULL, hInst, NULL );
    if (!hWnd)
        return (FALSE);

    ShowWindow( hWnd, numCmdShow );
    hContext = GetDC( hWnd );                   /* Need Display Context */

    /******************* Call Device & Wait for Messages ***************/

    if (PipeSetup() == FALSE)
    {
        lpBuffer = "Cannot Find Pipe.";
        TextOut( hContext, 0, 0, lpBuffer, lstrlen(lpBuffer) );
    }
    else
        PipeRegisterCallback(hWnd, CallbackProc);   /* Tell Pipe Who to Call */

    ReleaseDC( hWnd, hContext );                /* Free Display Context */
    UpdateWindow( hWnd );

    while (GetMessage(&msg, NULL, NULL, NULL))  /* Typical Message Loop */
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return (msg.wParam);
}

/***************** Receives Call From Virtual Device Pipe *************/

void FAR PASCAL CallbackProc(HWND hWnd, unsigned message, WORD wParam, LONG lParam)
{
    PostMessage(hWnd, message, wParam, lParam); /* Can Only PostMessage() */
    return;                                     /* System State Is Unknown */
}
```

Listing 6.9 *continued*

```
/**************** Receives Posted Message From CallbackProc() *********/

long FAR PASCAL WindowProc(HWND hWnd, unsigned message, WORD wParam, LONG lParam)
{
    struct rmp
    {
        int BusyBit;
        char Buffer[128];
    } far *realModePtr = (struct rmp far *) lParam;  /* PMode ptr to RMode String */
    HDC       hContext;

    switch (message)
    {
        case WM_DESTROY:
            PostQuitMessage( NULL );
            PipeUnregisterCallback();
            break;
        case WM_USER:
            hContext = GetDC( hWnd );           /* Need Display Context */
            TextOut( hContext, 0, 0, realModePtr->Buffer, lstrlen(realModePtr->Buffer) );
            realModePtr->BusyBit = FALSE;
            ReleaseDC( hWnd, hContext );        /* Free Display Context */
            UpdateWindow( hWnd );
            break;
        default:
            return (DefWindowProc(hWnd, message, wParam, lParam));
    }
}

/* End of File */
```

Listing 6.10 pipefunc.c

```
#include <dos.h>
#include "windows.h"
/*
******************************************************************
Title:      PIPEFUNC.C - General Purpose Pipe Functions
Author:     Thomas W. Olsen
Version:    3.0
Compiler:   Microsoft C 6.0
            cl /AL /Gs /Zi /c PIPEFUNC.C                    for DOS
            cl /AL /Gsw /W3 /Oas /Zpe /Zi /c PIPEFUNC.C     for WINDOWS
******************************************************************
*/
#define GET_API_ENTRY_POINT             0x1684
#define DEVICE_ID                       0x2020
#define DEVICE_VERSION                  0x0300
#define GET_VERSION                     0
#define REGISTER_CALLBACK               1
#define UNREGISTER_CALLBACK             2
#define ALLOCATE_GDT_SELECTOR           3
#define FREE_GDT_SELECTOR               4
#define CALL_WINDOWS_PROCEDURE          5

FARPROC lpPipeProc = NULL;

BOOL PASCAL PipeSetup(void)
{
    BOOL retVal = FALSE;
    union REGS regs;
    struct SREGS sregs;

    segread(&sregs);                        /* Is The Device Loaded? */
    regs.x.ax = GET_API_ENTRY_POINT;
    regs.x.bx = DEVICE_ID;
    sregs.es  = regs.x.di = 0;
    int86x( 0x2F, &regs, &regs, &sregs);

    FP_SEG(lpPipeProc) = sregs.es;
    FP_OFF(lpPipeProc) = regs.x.di;

    if (lpPipeProc)                         /* Device Is Loaded */
    {
        _asm mov ax, GET_VERSION;           /* Get Version */
        retVal = (*lpPipeProc)() < DEVICE_VERSION ? FALSE:TRUE;
    }
    return(retVal);
}
```

Listing 6.10 *continued*

```
BOOL PASCAL PipeRegisterCallback(HWND hWnd, FARPROC lpFuncPtr)
{
    BOOL retVal = FALSE;

    if (lpPipeProc)
    {
        _asm mov ax, REGISTER_CALLBACK;
        _asm mov bx, hWnd;
        _asm mov cx, word ptr lpFuncPtr+2;
        _asm mov dx, word ptr lpFuncPtr;
        (*lpPipeProc)();
        retVal = TRUE;
    }
    return(retVal);
}

BOOL PASCAL PipeUnregisterCallback(void)
{
    BOOL retVal = FALSE;

    if (lpPipeProc)
    {
        _asm mov ax, UNREGISTER_CALLBACK;
        (*lpPipeProc)();
        retVal = TRUE;
    }
    return(retVal);
}

WORD PASCAL PipeAllocateSelector(WORD segment)
{
    _asm mov ax, ALLOCATE_GDT_SELECTOR
    _asm mov bx, segment
    return((*lpPipeProc)());
}

void PASCAL PipeFreeSelector(WORD selector)
{
    _asm mov ax, FREE_GDT_SELECTOR
    _asm mov bx, selector
    (*lpPipeProc)();
}
```

Listing 6.10 *continued*

```
BOOL PASCAL PipeCallWindowsProc(WORD message, WORD wParam, LONG lParam)
{
    BOOL retVal = FALSE;

    if (lpPipeProc)
    {
        _asm mov ax, CALL_WINDOWS_PROCEDURE
        _asm mov bx, message
        _asm mov si, wParam
        _asm mov cx, word ptr lParam+2
        _asm mov dx, word ptr lParam
        (*lpPipeProc)();
        retVal = TRUE;
    }
    return(retVal);
}

/* End of File */
```

Chapter 7

A Swapping Replacement for the `spawn()` *Family*

Ralf Brown

Since v2.00, MS-DOS has provided the capability for one program to invoke another. This capability, through the `EXEC` function (`INT 21h AX=4B00h`), has always had a limitation: both the calling program and the new "child" program must fit into memory at the same time, because DOS keeps the entirety of the calling program in memory. Since the calling program cannot continue execution until the child program terminates, an obvious solution to the problem of insufficient memory is to temporarily remove as much of the calling program as possible from memory. A number of libraries have been developed which do just that. This chapter discusses one such library, called `SPAWNO`, which leaves less than 500 bytes of the calling program in memory and provides a direct drop-in replacement for the standard C library's `spawn..()` family of functions. The code disk included with this book contains the full source code to `SPAWNO`.

A swapping `spawn()` is quite simple conceptually: before actually calling the DOS `EXEC` function, you must copy the program's memory to a safe place outside the DOS memory area and then free the copied memory. When the child program exits, you must restore the memory and free the resources used to temporarily store the program. The complexity lies in such details as minimizing the size of the resident code and determining which memory to copy — but there DOS helps out by allowing us to walk its memory chain.

The MS-DOS memory chain is a linked list of Memory Control Blocks (MCBs) immediately preceding each block of allocated memory. An MCB (see Table 7.1) contains three fields of interest:

- a flag indicating whether it is the last MCB in the chain,
- the size of the memory block associated with the MCB,
- the memory block's owner.

0	1	2	3	4	5	6	7
type	owner		size		XXXX	XXXX	XXXX
DOS v4+ Program Name (if PSP memory block)							
8	9	A	B	C	D	E	F

Offset	**Size**	**Description**
`00h`	BYTE	block type: `5Ah` if last block in chain, otherwise `4Dh`
`01h`	WORD	PJP segment of owner or `0000h` if free `0008h` if belongs to DOS
`03h`	WORD	size of memory in paragraphs
`05h`	3 BYTEs	not used by MS-DOS
DOS v4+		
`08h`	8 BYTEs	ASCII program name if PSP memory block or DR-DOS UMB, else garbage null-terminated if less than 8 characters

Table 7.1 ***Format and description of a Memory Control Block.***

The next MCB in the chain always follows immediately after the end of the current block. Thus, given the address of the first MCB, you can simply keep adding the size of the current memory block to get to the next MCB until the type flag indicates that there are no more MCBs. The format of MCBs and the memory chain were undocumented prior to the release of MS-DOS v5.0. Now that they have been documented by Microsoft, they may be used with a considerable degree of confidence that they will not change in future versions of DOS.

To actually walk the complete memory chain you must find the first MCB, because the memory chain is a one-way linked list. You can find the first MCB with an undocumented but widely-known MS-DOS call, "Get List of Lists" (*INT 21h AH=52h*). The List of Lists is an internal data structure through which the majority of DOS's internal data structures can be reached. The only data structure required by *SPAWNO* is the memory chain, whose beginning address is stored two bytes prior to the address returned in *ES:BX* by the DOS call. Therefore, the code to set *ES* to point at the first MCB is

```
mov     ah,52h
int     21h                     ; get list of lists
mov     es,es:[bx-2]            ; get address of first MCB
                                ;     into ES
```

The following snippet of code shows how *SPAWNO* walks the memory chain initially to determine how many blocks need to be processed and how much space will be required to store the program. As will be seen, *SPAWNO* walks the chain a number of times. The three fields of the MCB mentioned previously are all accessed by this code. The "last" flag is the byte at offset *0*, the owner is stored in the word at offset *1*, and the size is stored in the word at offset *3*.

```
; ES initially points at first MCB in chain
        xor     bx,bx                   ; initially no extra memory blocks
        mov     cx,_psp@
```

```
extra_count_loop:
        inc     bx                      ; we have another block of memory
        inc     dx                      ; always swap at least the MCB
        mov     ax,es
        inc     ax                      ; memory block follows MCB
        cmp     ax,cx                   ; is this our PSP memory block?
        je      extra_count_loop_end    ; if yes, skip
        mov     ax,es:[1]               ; check block's owner
        cmp     ax,cx                   ; do we own the block?
        jne     extra_count_loop_end    ; if not, go immediately to next
        add     dx,es:[3]               ; is ours, so swap out the block
extra_count_loop_end:
```

00h	*CDh*	*20h*	memory limit		XXXX	CP/M CALL 5		
08h	service request		stored *INT 22* termin. addr.				stored ^Break	
10h	*INT 23h* addr.		stored *INT 24* crit-err. addr.				parent PSP	
18h	DOS v2+ Job File Table							
20h								
28h					env segment		temporary SP	
30h	temporary SS		JFT size		Job File Table address			
38h	SHARE pointer to prev. PSP				XXXX	APPEND	NetWare use	
40h	DOS version		MS Windows v3		XXXX	XXXX	XXXX	XXXX
48h	flag	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX
50h	far-call DOS entry			XXXX	XXXX	available to make		
58h	first FCB an extended FCB				First default FCB			
60h								
68h					Second default FCB			
70h								
78h					XXXX	XXXX	XXXX	XXXX
80h	len	commandline (through offset FFH)						

Table 7.2 ***Format of the Program Segment Prefix.***

```
        cmp     byte ptr es:[0],'Z'    ; was this the last memory block?
        je      extra_count_done
        mov     ax,es
        add     ax,es:[3]              ; move to next MCB
        inc     ax
        mov     es,ax
        jmp     extra_count_loop
    extra_count_done:
```

The previous example skips the caller's Program Segment Prefix (PSP, see Table 7.2) memory block because the PSP has already been handled separately, since part of that main memory block will remain in memory to hold the `SPAWNO` stub. The loop moves from one MCB to the next by adding one (to account for the MCB proper) plus the memory block's size to the current MCB's segment address (unless it is already on the final MCB, which is indicated by the ASCII character `Z` (value

Offset	**Size**	**Description**
`00h`	2 BYTEs	`INT 20` instruction for CP/M CALL 0 program termination
`02h`	WORD	segment of first byte beyond memory allocated to program
`04h`	BYTE	unused filler
`05h`	BYTE	CP/M CALL 5 service request (`FAR JMP` to `000C0h`)
`06h`	WORD	CP/M compatibility — size of first segment for `.COM` files
`08h`	2 BYTEs	remainder of `FAR JMP` at `05h`
`0Ah`	DWORD	stored `INT 22` termination address
`0Eh`	DWORD	stored `INT 23` control-Break handler address
`12h`	DWORD	DOS v1.1+ stored `INT 24` critical error handler address
`16h`	WORD	segment of parent PSP
`18h`	20BYTEs	DOS v2+ Job File Table, one byte per file handle, FFH = closed

Table 7.2 ***continued — Descriptions of PSP offsets.***

5Ah) in the first byte of the MCB). Each time through the loop, the count of memory blocks is incremented. The total number of paragraphs to be swapped out is incremented to account for the MCB (which is always swapped out) and then further increased by the size of the memory block in paragraphs if it belongs to the caller.

Once *SPAWNO* knows how much storage will be required to swap out the program, it attempts to allocate that amount of XMS or EMS. Should these attempts fail, it falls back to using disk space as temporary storage and creates a swap file. In any case, *SPAWNO* remembers which storage

Offset	Size	Description
2Ch	WORD	DOS v2+ segment of environment for process
2Eh	DWORD	DOS v2+ process's *SS:SP* on entry to last *INT 21* call
32h	WORD	DOS v3+ number of entries in JFT (default 20)
34h	DWORD	DOS v3+ pointer to JFT (default *PSP:0018h*)
38h	DWORD	DOS v3+ pointer to previous PSP (default *FFFFFFFFh* in 3.x)
3Ch	BYTE	apparently unused by DOS versions through v6.20
3Dh	BYTE	(APPEND) TrueName flag (see *INT 2F/AX+B711h*)
3Eh	BYTE	(Novell NetWare) flag: next byte initialized if *CEh*
3Fh	BYTE	(Novell NetWare) Novell task number if previous byte is *CEh*
40h	2 BYTEs	DOS v5+ to return on *INT 21/AH=30h*
42h	WORD	(MSWindows3) selector of next PSP (PDB) in linked list Windows keeps a linked list of Windows programs only
44h	4 BYTEs	unused by DOS versions through 6.20
48h	BYTE	(MSWindows3) bit 0 set if non-Windows application (WINOLDAP)
49h	7 BYTEs	unused by DOS versions through 6.20

Table 7.2 *continued*

method is being used for swapping. For both XMS and EMS, *SPAWNO* first checks whether the appropriate driver is installed, and if so, how much memory is available for allocation.

The installation check for an XMS driver is *INT 2Fh AX=4300h* (see the code following *check_XMS* in *__SPAWNV.ASM*). If this call returns *AL* set to *80h*, we may call *INT 2Fh AX=4310h* to get the driver's entry point into *ES:BX*. All further calls to the XMS driver will be invoked through a far call to this entry point. XMS function *08h* returns the size of the largest available block (in Kb) in register *AX*, while function *09h* attempts to

Offset	**Size**	**Description**
50h	3 BYTEs	DOS v2+ service request (*INT 21/RETF* instructions)
53h	2 BYTEs	unused in DOS versions through 6.20
55h	7 BYTEs	unused in DOS versions through 6.00; can be used to make first FCB into an extended FCB
5Ch	16BYTEs	first default FCB, filled in from first commandline argument; overwrites second FCB if opened.
6Ch	16BYTEs	second default FCB, filled in from second commandline argument; overwrites beginning of commandline if opened
7Ch	4 BYTEs	unused
80h	128BYTEs	commandline / default Disk Transfer Address (DTA) command tail is BYTE for length of tail, N BYTEs for the tail, followed by a BYTE containing *0Dh*

Table 7.2 *continued*

allocate a block whose size is specified by `DX`. If function `08h` indicates insufficient memory or function `09h` is unable to allocate the requested amount (which may occur under a multitasker if some other program was given a time slice between the two XMS calls), `SPAWNO` falls back to attempting EMS.

The installation check for an EMS driver is slightly more complicated and is shown in function `_EMS_available`. First, the installation check code retrieves the interrupt `67h` handler's address, and then it tests for the presence of the string `EMMXXXX0` at offset `10` in the handler's segment. If the string is not present, then EMS memory is not available — either because no driver is present or it has been disabled — so the amount of EMS memory available is zero pages. If the string matches, however, an EMS driver is present and enabled, and `INT 67h AH=42h` is called to retrieve the number of pages available for allocation.

Should there be insufficient XMS or EMS memory available for swap space, `SPAWNO` falls back to using disk space for swapping. It first copies the user-specified swap directory into an internal buffer and then appends the filename `OVRLnnnn.$$$`, where "`nnnn`" is the four-digit hex number representing the program's PSP segment address. (DOS uses the PSP segment address as a process identifier. It is unique within the machine at any given time.) Finally, `SPAWNO` attempts to create the file and, if the creation is successful, closes and reopens the file with file sharing modes set to prevent alteration of the file by other processes. Should the file creation fail, `SPAWNO` returns immediately with an error code.

Once the necessary swap space has been allocated, the next step is to copy the stub which will reload the program as low in the main memory block as possible; this will minimize the size of the program's resident portion. In doing so, `SPAWNO` actually builds its resident code out of two pieces: a common section which sets up the registers and resizes the main memory block, and a swap-device specific section which actually reads back the program's main memory block. After that, `SPAWNO` switches to a local stack located in the resident stub and writes out the program's main memory block. The stack is switched this early so that the swap space will contain a correct copy of the program's stack.

Writing out the remaining memory blocks requires a second sweep through the memory chain. This time, instead of merely noting the block's size, the program copies the block's contents to the swap space if the block belongs to the program rather than someone else. In examining the function *write_other_blocks*, you will notice that every MCB is written, regardless of whether the memory block itself is written. This preserves an image of the entire memory chain which *SPAWNO* will follow when it restores the program's memory later. After copying the program's memory, *write_other_blocks* walks the memory chain another time to free all blocks belonging to the program except the main PSP block, which will merely be resized.

The main PSP memory block was written first to simplify the code and thus make it smaller. Since the main block was written first, we must reset the file pointer to the beginning of the file if we are swapping to disk, so that the resident stub will read the proper data. To ensure that all data has been written to disk and that there are no problems from programs like *CHKDSK*, *SPAWNO* commits the file by making a duplicate of the file handle and closing the duplicate, as shown below. This forces DOS to flush any buffers associated with the file and update the file's directory entry. If the file had not been committed in this fashion, the directory entry would record a length of zero bytes, and *CHKDSK* would happily truncate the swap file.

```
commit_swap_file:
        mov     ah,45h          ; make a duplicate of our file handle
        int     21h
        jc      do_overlay      ; but make sure not to close only copy
        xchg    bx,ax           ; put duplicate handle in BX
        push    ax              ; preserve original file handle
        mov     ah,3Eh          ; close duplicate file handle
        int     21h
        pop     bx              ; get back original file handle
do_overlay:
```

At this point (*do_overlay* in the code on disk), the only remaining actions are to unhook interrupts *23h* (Ctrl-Break) and *24h* (Critical Error), shrink the main memory block, store the data needed to reload the main memory block on the stack, and set up the registers for invoking the DOS *EXEC* function. We then jump to the resident stub, where the very first instruction (at *$overlay* in the source code) is the *INT 21h* which actually invokes the *EXEC* function.

On return from the *EXEC*, the stack pointer is restored, since PC-DOS v2.x has destroyed it, and the swap-in data is popped off the stack into appropriate registers. The returned carry flag and any error code are then pushed to preserve them until we return to the main *SPAWNO* code.

SPAWNO expands the main memory block to its original size before attempting to restore its contents. Should the resizing fail, there is no recourse but to display an error message and abort the program, because some other program (most likely a TSR installed by the child program) is using the memory which will be required to swap the caller back into memory. If the resizing is successful, *SPAWNO* can then read in the main memory block, which it does with one of three routines (one for each type of temporary storage). The appropriate routine was appended to the common resident code when the stub was copied to its current location near the beginning of the memory block. The resident stub need only contain enough code to load in the main memory block, since doing so restores the remainder of *SPAWNO*'s code. Thus, *SPAWNO* bootstraps itself and the program back into memory: the resident stub reads in the rest of the program, including the main *SPAWNO* code, which in turn reads in all the other memory blocks owned by the program which had been swapped out of memory.

After the resident stub has reloaded the main memory block, it executes a far return back to the main *SPAWNO* code, which stores the result of the *EXEC* call, retrieves the child program's exit code if the *EXEC* was successful, and starts restoring everything to the way it was before *SPAWNO* was called. This involves switching the stack back from the local stack to the original stack, restoring the portion of low memory overwritten by the resident stub, restoring *INT 23h* and *INT 24h*, reading in all other memory blocks which were swapped out, and freeing the temporary storage used as swap space.

When reading in the other memory blocks, *SPAWNO* follows a virtual memory chain stored in the swap space. After retrieving the address of the first MCB from DOS, it reads the first MCB stored in the swap space into a temporary buffer and checks its owner. If the block was originally owned by the caller, we read in the remainder of the memory block; if it was free, we only copy the MCB from the temporary buffer to its original location; if it belonged to someone else, we skip it entirely. The MCB in the temporary buffer is then consulted to determine the address of the next MCB, and the process is repeated until all MCBs and any associated memory blocks have been restored.

After restoring all memory, *INT 23h*, and *INT 24h*, and freeing the temporary storage, *SPAWNO* retrieves the return code which was placed on the stack just after returning from the child program and returns to its caller.

In order to leave as little of itself in memory as possible, *SPAWNO* resorts to some tricks that minimize the size of its resident stub. One method has already been touched on indirectly. Instead of reloading all memory blocks, the resident stub can read in only the single memory block containing the remainder of the program's (and *SPAWNO*'s) code. The memory space can be further reduced if you include only the code needed for the particular kind of swap space being used by the current invocation.

A second method of minimizing the resident stub's size is to re-use as much of the PSP as possible. On examining the layout of the PSP, one can see that the program's command-line area could be re-used, since it is not needed for anything by DOS (only by the application, and *SPAWNO* will restore any memory it overwrites; the application will never be aware that it had been used for something else). Similarly for the two default File Control Blocks (FCBs), the far-call DOS entry point at offset *50h*, and even the DOS v5+ *SETVER* at offset *40h*. Thus, we can safely re-use fully three-fourths of the PSP, from offset *40h* to the end at offset *FFh*. Examining the definition of *PSP_SEG* in *__SPAWNV.ASM* shows that this is precisely what *SPAWNO* does.

The third method is even sneakier: place some of the data to be used by the *EXEC* in unallocated memory which will eventually be overwritten by the *EXEC* code or the child program — but not before it has been used. The two items placed in unprotected memory are the child program's filename and the environment to be copied for the child. The DOS *EXEC* code opens the executable to be loaded as very nearly its first action and never requires the filename thereafter. Similarly, a copy of the environment is made before the child program is read into memory, and the environment block from which the copy was made is also never used again. Note that this approach has a potential incompatibility with certain resident anti-viral programs which swap their scanning code into unallocated memory on an *EXEC* call. But the problem of the anti-viral overwriting the program name or environment will only occur for small programs which don't gain much from *SPAWNO* anyway.

It should come as no surprise that a fair amount of the total *SPAWNO* code is dedicated to ensuring compatibility with the standard library's *spawn()* functions. Thus, for example, all of the separately-specified arguments for the child program are concatenated into a DOS command line which can be passed to the *EXEC* function. The C-style environment string array must be converted into a DOS environment block. And many of the *spawn()* functions call for the ability to search the *PATH* for the executable.

The child program's arguments are concatenated into a DOS command line beginning near *parm_cat_loop* in *__SPAWNV.ASM*. All of the replacement *spawn..()* functions pass the actual spawning function *__SPAWNV* a pointer to an array of pointers to argument strings, whether the *spawn..()* function takes a list or an array. The first item in the array is *argv[0]*, which is ignored under DOS, and the last item is signalled by a *NULL* pointer. For each item between those two, the code between *another_parm* and *parm_copy_done* appends a blank to the already-produced command line, then appends the argument string, keeping track of how large the command line has become. Should the full 126 characters (128 bytes less the initial length byte and trailing carriage return) available for the command line be used up, the copying is terminated immediately. After building the command line, the two default FCBs for the child program are built using *INT 21h AH=29h* to parse the beginning of the command line into each of the two FCBs. Few programs actually use these FCBs nowadays, but the DOS *EXEC* function expects to receive them for backward compatibility with old DOS v1.x programs.

The environment string array is converted to a DOS environment block by the function *__spawn_buildenv()* in *SPAWNENV.C*. This function loops through the array twice, the first time to determine the size of the environment block which will be created. The end of the array is signalled by a *NULL* pointer or an empty string:

```
for (i = 0 ; env[i] && *env[i] ; i++) /* add up the lengths of the */
   length += strlen(env[i]) + 1 ;     /* environment variables */
```

Once the required length has been determined, we can allocate a block of memory to hold the new environment. Since there is no guarantee that the C *malloc()* function will return a paragraph-aligned block, as required by DOS, we allocate an additional fifteen bytes and compute the first segment address following the beginning of the returned block. This results in a block of the desired size which is paragraph aligned, into which we can then concatenate all the environment strings:

```
mem = (char far *)malloc(length+15) ;
seg = FP_SEG(mem) + ((FP_OFF(mem)+15)>>4) ;
```

The *spawn?p?()* functions require the ability to scan the search path defined by the *PATH* environment variable; this capability is implemented by *__SPAWN_SEARCH* in the file *SPAWNPTH.ASM*. *__SPAWN_SEARCH* first finds the *PATH* variable in the environment by retrieving the segment address of the environment from the PSP and then searching the environment for "*PATH=*." For each component of the search path, it copies the directory name into an internal buffer, appends the desired executable's name, and calls the function *CHECK_DIR*. *CHECK_DIR* appends "*.**" if no extension is given in the specified filename and then uses the DOS *FindFirst* and *FindNext* functions (*INT 21h AH=4Eh* and *AH=4Fh*) to test for the existence of the specified filename or the given name with extension *.COM* or *.EXE*. Like *COMMAND.COM*, *CHECK_DIR* gives precedence to *.COM* if it finds both a *.COM* and a *.EXE* file with the given base name. Once a matching file is found, *CHECK_DIR* returns success (changing the extension stored in the internal buffer as appropriate), and *__SPAWN_SEARCH* returns that string to its caller.

Swapping a program out of memory while shelling out to another program is conceptually straight-forward; the devil lies in the details, both in the actual swapping and in providing compatibility with the standard run-time libraries. Although *SPAWNO* makes only one undocumented call, it does rely on undocumented details of the behavior of the *EXEC* function in order to further reduce its footprint.

The disk accompanying this book contains the full source code to *SPAWNO*, as well as pre-compiled libraries and documentation. Instructions for fine-tuning *SPAWNO* in your programs are in *SPAWNO.DOC*, but for many uses, simply linking the appropriate model of *SPAWNx.LIB* into your existing code will suffice to convert your application into one which swaps itself out of memory while shelling out to another program.

Chapter 8

Exploiting the 386 Four-byte Moves

Davie Lee Reed

The 8088 by design cannot address more than 1Mb of RAM. This limitation forces 8088 users to use bank switched memory, such as EMS *expanded* memory, to access more than 1Mb of RAM. The EMS expanded memory is accessed through multiple 16Kb blocks of RAM which is addressed within the 1Mb range. The 80286 overcame this major problem by allowing the addressing of up to 16Mb of RAM. The RAM above the one megabyte address is called *extended* memory. This extended memory can only be accessed by switching the 80286 into protected mode, unless you use the undocumented `LOADALL` instruction. (See Chapter 9.)

Most applications that use any of the additional 15Mb do so for storing data, such as a RAM-DISK. Most of these programs use the BIOS `INT 15H` service to switch the 80286 CPU into protected mode and transfer data to and from extended memory. Once the data is transferred, the BIOS also switches the 80286 back into *real mode* (the power-up default).

Unfortunately, the 80286 has no instruction that switches back to real mode. Thus the BIOS is forced to resort to a kludge — it immediately resets the 80286. The BIOS avoids a cold-start after this reset (which would just place the user back at the DOS prompt) by saving the current context (the current instruction pointer, flags, etc.) into the CMOS RAM. Since the CMOS RAM is *not* cleared upon boot-up, the BIOS initialization code can test the CMOS for a valid context and if it finds one, restart the 80286 at the BIOS location which continues the extended memory transfer. While this kludge works, all of the fooling around with CMOS, resetting the CPU, and special boot-up checking makes the operation of moving memory slower.

The 80386 solves this problem. The 386 can access any of the 15Mb of extended memory and does not require a reset to return to real mode. This makes extended memory operations much faster. In addition, the 80386 can move four memory bytes at a time — the 80286 and 8088 can move only two bytes at a time. In this chapter I'll show you how to detect the presence of a 80386 CPU so that your program can exploit its quadbyte moves.

Detecting the 386

Listing 8.1 will detect the following CPUs: 8088, 8086, 80186, 80286, 80386, NEC V20 and NEC V30.

To work with the other code, Listing 8.1 should be named `CPUSTUFF.ASM` and should be assembled using MASM or a MASM-compatible assembler to produce a file called `CPUSTUFF.OBJ`. Listing 8.2 shows how to call the detection routine in assembly language.

`CHIPTYPE` is an `EXTERNAL` module that must be linked in at link time. The `PUSH`es place the address of the variable `CPUTYPE` onto the stack so that `CHIPTYPE` knows where to store the answer. The possible returned values for `CPUTYPE` are:

```
0:8088
1:8086
2:NEC V20
3:NEC V30
4:80188
5:80186
6:80386
7:80286
```

`CHIPTYPE` can also be linked into a Turbo Pascal (version 4.0 and up) program. The `CPUSTUFF.OBJ` is linked into the Turbo Pascal program as a `PROCEDURE`. Listing 8.3 shows how `CHIPTYPE` would be invoked from Pascal.

In Listing 8.3, `CPUSTUFF.OBJ` has been linked as a `NEAR` (not `FAR`) procedure type called `ChipType`. `ChipType` is then called in the main program logic to retrieve the CPU type into the variable `CpuVal` which is then printed on the video display.

Equipped with this detection routine, you can write block move routines that adapt to the capabilities of the CPU. Listing 8.4 shows how to check for the presence of the 386 using our previous routines and then copy memory using the appropriate `MOV` instructions. The source code is in assembler language and must be assembled using Turbo Assembler. If you do not have Turbo Assembler, you may change the CPU directives (`.8086` and `.386`) in the source code to be compatible with your assembler.

For the move routine in Listing 8.4 to work properly, the variable `CPUTYPE` must have been initialized by calling the routine `CHIPTYPE`.

I hope that the routines given in this article will motivate you to rewrite portions of your code to take advantage of 386 machines so that they will perform even faster.

[Editor's note: We include this Chapter in the 3rd Edition because we applaud the practice of programming for the newer chips but maintaining compatibility with the older chips. This method checks for the 80386 and the NEC V 80486. See Chapter 9 (Listing 9.4) for an alternative method that checks for the 80486.]

Listing 8.1 CPUSTUFF.ASM

```
.***************** PROGRAM SECTION START ******************
;
; CHECK THE PROCESSOR TYPE!
;
; 0 = 8088
; 1 = 8086
; 2 = NEC V20
; 3 = NEC V30
; 4 = 80188
; 5 = 80186
; 6 = 80386
; 7 = 80286
;
        PUBLIC  CHIPTYPE

CSEG    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  DS:CSEG,CS:CSEG
CHIPTYPE PROC   NEAR
;
; This routine will take the CPU type and store it at
; the address placed on the stack. The routine is setup
; to work hand in hand with TURBO PASCAL 4.0 & up.
;
PARM1   EQU     6       ;SEG
PARM2   EQU     4       ;DISP
        PUSH    BP
        MOV     BP,SP

        PUSH    ES
        PUSH    DS
LCPU:
        MOV     BX,6
        PUSH    SP
        POP     AX
        CMP     AX,SP
        JNZ     TST88V
;
; THE 286 & 386 CPU PUSHES THE VALUE OF THE SP BEFORE DECREMENTING SP
; THE PRE-286 CPU PUSHES THE SP AFTER DECREMENTING SP
;
; 286 or 386 ?
;
        PUSHF
        XOR     AX,AX
        MOV     AX,0F000H
        PUSH    AX
        POPF
        PUSHF
        POP     AX
        AND     AX,0F000H
        JNZ     YES386
        POPF
        JMP     TSTQUE
```

Listing 8.1 *continued*

```
;
; Definitely 386 because the upper 4 bits of the FLAGS operate
; differently on a 386
;
YES386: POPF
       MOV     BX,66H
       JMP     IS286
;
; 8088 or NEC V series chip?
;
TST88V: MOV     BX,4
       MOV     CL,21H
       MOV     AL,0FFH  ;Can I shift a lot?
       SHL     AL,CL
       JNZ     TSTQUE   ;If so then NOT a 80188 or 80186
       MOV     BX,2     ;otherwise could be V20, V30, 8088 or 8086
       CALL    ISVSERIES ;If CX=0 then either V20 or V30
       JCXZ    TSTQUE
       MOV     BX,0     ;Has to be 8088 or 8086
TSTQUE: CALL    BIGQUE   ;If pre-fetch QUE BIG then must be V30 or 86
       JCXZ    IS286
       OR      BX,1
IS286:  CMP     BX,6     ;If 286 type then force to 7
       JNZ     L286
       MOV     BX,7
L286:   AND     BX,07H
       STI
       MOV     DS,[BP+PARM1]
       MOV     SI,[BP+PARM2]
       MOV     DS:[SI],BX
       POP     DS
       POP     ES
       MOV     SP,BP
       POP     BP
       RET     4              ;POP PARMS OFF STACK
CHIPTYPE ENDP
BIGCUE  PROC    NEAR
```

Listing 8.1 *continued*

```
;
; CHECK FOR 8086 OR 80186
;
; TESTS THE PROCESSOR'S PRE-FETCH QUEUE
;
        PUSH    CS
        POP     DS
        PUSH    CS
        POP     ES
        STD
        CALL    $+3
        POP     DI
        ADD     DI,OFFSET PATCH-$+1
        MOV     AL,BYTE PTR CS:[DI-1]
        CLI
        MOV     CX,3
        REPZ    STOSB
        CLD
        NOP
        NOP
        NOP
        INC     CX
CHKPNT: STI
PATCH:  STI
        MOV     BYTE PTR DS:[DI+1],41H
        RET
BIGCUE  ENDP
ISVSERIES PROC    NEAR
;
; COULD RETURN CX<>0 BECAUSE OF SEGMENT PREFIX IF NOT A 386 V20 OR V30
;
; IF NOT A V20 OR V30 AN INTERRUPT WILL CAUSE THE INSTRUCTION
;    NOT COMPLETE THUS YIELDING CX<>0
;
        MOV     CX,0FFFFH
        STI                     ;ENABLE INTERRUPTS
        PUSH    AX
        PUSH    SI
        DB      0F3H,026H,0ACH  ;REP ES:LODSB
        POP     SI
        POP     AX
        RET
ISVSERIES ENDP
CSEG    ENDS
        END
.***************** PROGRAM SECTION END ******************

; End of File
```

Listing 8.2 TESTCALL.ASM

```
.****************** PROGRAM SECTION START ******************
.
       EXTRN   CHIPTYPE:NEAR
.
CPUTYPE DW      0       ;CPU TYPE VALUES, 0-7.
.
       PUSH    CS
       MOV     AX,OFFSET CPUTYPE
       PUSH    AX
       CALL    CHIPTYPE                ;GET CPU TYPE STORED IN CPUTYPE
.
.****************** PROGRAM SECTION END ******************

. End of File
```

Listing 8.3 CPUCHECK.PAS

```
.****************** PROGRAM SECTION START ******************
.
{$F-}    {Far Calls Turned OFF!}
{$L CPUSTUFF}
Procedure ChipType(Var X:Integer);
External;
.
Var
  CpuVal:Word;.
Begin
ChipType(CpuVal);
Writeln(’This CPU’s value is:’,CpuVal);
End.
.****************** PROGRAM SECTION END ******************

. End of File
```

Listing 8.4 DOMOVES.ASM

```
.***************** PROGRAM SECTION START ******************
.
DOMOVES PROC    NEAR
;
; THIS ROUTINE WILL TAKE THE PLACE OF ANY (REP MOVSW) PROCESS.
; BY CALLING THIS ROUTINE INSTEAD OF USING (REP MOVSW) INSTRUCTIONS, YOU
; WILL INCREASE THE SPEED OF MEMORY MOVING SIGNIFICANTLY ON 386 AND 486
; CPU BASED MACHINES.
;
; SINCE THIS IS A (REP MOVSW) REPLACEMENT IT STANDS TO REASON THAT ANY
; AND ALL REQUIREMENTS OF THE (REP MOVSW) INSTRUCTION ARE STILL
; REQUIRED. FOR EXAMPLE:
;
;       DS:SI     POINTS TO SOURCE MEMORY
;       ES:DI     POINTS TO DESTINATION MEMORY
;       CX        NUMBER OF WORDS TO MOVE
;
; SHOULD I DO 386 (32 BIT MOVES?)
;
        CMP     DS:CPUTYPE,6    ;AM I ON A 386?
        JZ      RPMX386B
        REP     MOVSW           ;NO JUST DO REGULAR REPEAT WORD MOVES
        JMP     NO386B          ;DONE
RPMX386B:
;
; IF 386 THEN MOVE AS MANY DOUBLE WORD MULTIPLES AS YOU CAN THEN
; IF THERE WAS AN ODD NUMBER OF WORDS TO MOVE, FINISH OUT WITH
; A FINAL WORD MOVE.
;
        SHR     CX,1            ;CX=HOW MANY DOUBLE WORDS TO MOVE
.386                            ;TELLS TURBO ASSEMBLER TO USE 386 OPCODES
        REP     MOVSD
.8086                           ;SWITCHES BACK TO 8088-8086 OPCODES
        JNC     NO386B          ;WAS THERE AN ODD NUMBER OF WORDS?
        MOV     CX,1
        REP     MOVSW           ;IF SO THEN FINISH OUT LAST MOVE
NO386B:
        RET
DOMOVES ENDP
.
.***************** PROGRAM SECTION END ******************

. End of File
```

Chapter 9

The LOADALL *Instruction*

Robert Collins

Of the few undocumented instructions in the 80286 and 80386 microprocessors, the *LOADALL* instruction is the most widely known. Nevertheless, very few people understand how to use it. Using *LOADALL* is not as simple as merely knowing the *LOADALL* opcode and its format, because knowing how to use *LOADALL* requires a knowledge of many aspects of the CPUs' behavior that are not documented in their respective data sheets.

The 286 *LOADALL* is widely known because a 15-page Intel confidential document describing its use was given to many developers. 286 *LOADALL* is so commonly used in production code that DOS v3.3 (and above) and OS/2 have provisions for using *LOADALL* built in them. Every 386 and 486 BIOS emulates 286 *LOADALL*, and even Microsoft *CODEVIEW* recognizes the 286 *LOADALL* opcode and disassembles it.

On the other hand, the 386 *LOADALL* is not widely known, and very few developers even know it exists. In this chapter, I will explain how to use both the 286 and 386 *LOADALL* instructions and present source code to demonstrate the various aspects of CPU behavior that become apparent, or can be proven, when using *LOADALL*.

Intel originally included *LOADALL* in the CPU mask for testing purposes and In Circuit Emulator (ICE) support. As its name implies, *LOADALL* loads all of the CPU registers, including the "hidden" software-invisible registers. At the completion of a *LOADALL* instruction, the entire CPU state is defined according to the *LOADALL* data table. *LOADALL* loads all of the software-visible registers such as *AX*, and all of the software-invisible registers such as the segment descriptor caches.

By manipulating the descriptor cache base registers, you can access the entire address space without switching to protected mode. In other words, by using *LOADALL*, you can access memory above 1Mb from real mode. Since the alternative method for the 286 (switching to protected mode, accessing the desired memory, then resetting the CPU — the only way to get the 286 back to real mode) has a significant performance penalty, *LOADALL* is most significant to 286 programmers. *LOADALL* provides them with a new capability that is not available by any other means.

LOADALL *Details*

LOADALL is closely coupled with the CPU hardware. Both the 286 and 386 have different internal hardware, and Intel implemented *LOADALL* using different opcodes on the 286 and 386. 80286 *LOADALL* (opcode *0F05*) produces an invalid opcode exception when executed on the 386, and 80386 *LOADALL* (opcode *0F07*) produces an invalid opcode exception when executed on the 286.

LOADALL loads all CPU registers (including *MSW*, *GDTR*, *CSBASE*, *ESACCESS*) from a memory image. You can execute *LOADALL* in real or protected mode, but only at privilege level *0 (CPL=0)*. If you execute *LOADALL* at any other privilege level, the CPU generates an exception.

By directly loading the descriptor cache registers with *LOADALL*, a program has explicit control over the base address, segment limit, and access rights associated with each memory segment. Normally, the CPU loads these values each time it loads a segment register, but *LOADALL* allows you to load these hidden registers independently of their segment register counterparts.

In real mode, *LOADALL* makes it possible to access a memory segment that is not associated with any segment register. Likewise in protected mode, you can access memory that has no descriptor table entry.

Physical Address	Description	Size	Value
[800]	None	DW	0
[902]	None	DW	0
[804]	None	DW	0
[806]	MSW	DW	?
[808]	None	DW	0
[80A]	None	DW	0
[80C]	None	DW	0
[80E]	None	DW	0
[810]	None	DW	0
[812]	None	DW	0
[814]	None	DW	0
[816]	TR_REG DW	?	
[818]	FLAGS DW	?	
[81A]	IP	DW	?
[81C]	LDT_REG DW	?	
[81E]	DS_REG DW	?	
[820]	SS_REG DW	?	
[822]	CS_REG DW	?	
[824]	ES_REG DW	?	
[826]	DI	DW	?
[828]	SI	DW	?
[82A]	BP	DW	?
[82C]	SP	DW	?
[8E0]	BP	DW	?
[830]	DX	DW	?
[832]	CX	DW	?
[834]	AX	DW	?
[836]	ES_DESC	DESC_CACHE286	<?,?,?>
[83C]	CS_DESC	DESC_CACHE286	<?,?,?>
[842]	SS_DESC	DESC_CACHE286	<?,?,?>
[848]	DS_DESC	DESC_CACHE286	<?,?,?>
[84E]	GDT_DESC	DESC_CACHE286	<0,?,?>
[854]	LDT_DESC	DESC_CACHE286	<?,?,?>
[85A]	IDT_DESC	DESC_CACHE286	<0,?,?>
[860]	TSS_DESC	DESC_CACHE286	<?,?,?>
[866]	END OF TABLE		
DESC_CACHE286 STRUC			
Addr_A15_A00		DW	?
Addr_A23_A16		DB	?
_Access		DB	?
_Limit		DW	?
ENDS			

Table 9.1 *80286* LOADALL *data structure format.*

LOADALL performs no protection checks against any of the loaded register values. When you execute it at *CPL 0*, *LOADALL* can generate no exceptions. The segment access rights and limit portions may be values that would otherwise be illegal in the context of real mode or protected mode, but *LOADALL* willingly loads these values with no checks. Once loaded, however, the CPU performs full access checks when accessing a segment. For example, you can load a segment whose access is marked "not present." Normally, this condition would generate exception 11, "segment not present," but *LOADALL* does not generate exception 11. Instead, any attempt to access this segment will generate exception 13.

LOADALL does not check coherency between the software-visible segment registers and the software-invisible segment descriptor cache registers. Any segment descriptor base register may point to any area in the CPU address space, while the software-visible segment register may contain any other arbitrary value. The CPU makes all memory references according to the descriptor cache registers, not the software-visible segment registers. All subsequent segment register loads will reload the descriptor cache register. Beware of using values in *CS* that do not perfectly match a code segment descriptor table entry, or a real mode code segment — an interrupt return (*IRET*) may either cause an exception or execution to resume at an unexpected location. Likewise, pushing and subsequently popping any segment register will force the descriptor cache register to reload according to the CPU's conventional protocol, thereby inhibiting any further real mode extended memory references.

Offset	Description
0-2	24-bit physical address of the segment in memory. These bytes are stored in standard Intel format with the least significant byte at the lowest memory address.
3	Access rights. The format of this byte is the same as that in the descriptor table. This access byte is loaded in the descriptor cache register regardless of its validity. Therefore the "present" bit in the access rights field becomes a "descriptor valid" bit. When this bit is cleared, the descriptor is considered invalid, and any memory reference using this descriptor generates exception 13, with error code 0. The Descriptor Privilege Level (DPL) of the SS and CS descriptor caches determines the Current Privilege Level (CPL). The CS descriptor cache may be loaded as a read/write data segment.
4-5	Segment limit. The standard 16-bit segment limit stored in standard Intel format.

Table 9.2a ***80286 descriptor cache entry formats.***

80286 LOADALL

You encode the 80286 *LOADALL* as a two-byte opcode, *0F05h*. *LOADALL* reads its table from a fixed memory location at *800h* (80:0 in real-mode addressing). *LOADALL* performs 51 bus cycles (*WORD* cycles) and takes 195 clocks with no wait states. Table 9.1 shows the format you must prepare at location *800h* before executing the 286 *LOADALL* instruction. All CPU register entries in the *LOADALL* table conform to the standard Intel format, where the least significant byte is at the lowest memory address. Table 9.2 shows the 286 format of the descriptor cache entries.

Intel recommends some guidelines for proper execution following *LOADALL*. The stack segment should be a read/write data segment; the code segment can be execute only (*access=95h*), read/execute (*access=9bh*), or read/write/execute (*access=93h*). Proper protected mode operation also requires that the DPL of *CS* and DPL of *SS* be equal. These attributes determine the CPL of the processor. Also, the DPL fields of *ES* and *DS* should be equal to *3* to prevent *RETF* or *IRET* instructions from zeroing these registers.

The code in Listing 9.1 demonstrates how to explore the various operating modes with 286 *LOADALL* and how to access extended memory while in real mode. The *LOADALL* test performs various functions that would be impossible to duplicate without using *LOADALL* (see Listing 9.4 for common code to detect the CPU type).

80386 LOADALL

The 386 *LOADALL* is encoded as a two-byte opcode (*0F07*). Unlike the 286 *LOADALL*, this *LOADALL* instruction reads its data from a table pointed to by *ES:EDI*. Segment overrides are allowed, but apparently ignored. The 386 *LOADALL* performs 51 bus cycles (*DWORD* cycles) and takes 122 clocks with

Offset	Description
0-2	24-bit physical address of the segment in memory.
3	Should be 0.
4-5	Segment limit.

Table 9.2b ***GDT and IDT descriptor cache entry formats.***

no wait states. Table 9.3 shows the 386 `LOADALL` format. However, Table 9.3 does not show that prior to reading the `LOADALL` table, `LOADALL` reads

Offset	Description	Size	Value
[00]	*CRO*	*DD*	?
[04]	*EFLAGS*	*DD*	?
[08]	*EIP*	*DD*	?
[0C]	*EDI*	*DD*	?
[10]	*ESI*	*DD*	?
[14]	*EBP*	*DD*	?
[18]	*ESP*	*DD*	?
[1C]	*EBX*	*DD*	?
[20]	*EDX*	*DD*	?
[24]	*ECX*	*DD*	?
[28]	*EAX*	*DD*	?
[2C]	*DR6*	*DD*	?
[30]	*DR7*	*DD*	?
[34]	*TR_REG*	*REG_STRUC*	<?>
[38]	*LDT_REG*	*REG_STRUC*	<?>
[3C]	*GS_REG*	*REG_STRUC*	<?>
[40]	*FS_REG*	*REG_STRUC*	<?>
[44]	*DS_REG*	*REG_STRUC*	<?>
[48]	*SS_REG*	*REG_STRUC*	<?>
[4C]	*CS_REG*	*REG_STRUC*	<?>
[50]	*ES_REG*	*REG_STRUC*	<?>
[54]	*TSS_DESC*	*DESC_CACHE*	<?,?,?>
[60]	*IDT_DESC*	*DESC_CACHE*	<0,?,?>
[6C]	*GDT_DESC*	*DESC_CACHE*	<0,?,?>
[78]	*LDT_DESC*	*DESC_CACHE*	<?,?,?>
[84]	*GS_DESC*	*DESC_CACHE*	<?,?,?>
[90]	*FS_DESC*	*DESC_CACHE*	<?,?,?>
[9C]	*DS_DESC*	*DESC_CACHE*	<?,?,?>
[A8]	*SS_DESC*	*DESC_CACHE*	<?,?,?>
[B4]	*CS_DESC*	*DESC_CACHE*	<?,?,?>
[C0]	*ES_DESC*	*DESC_CACHE*	<?,?,?>
[CC]	LENGTH OF TABLE		

REG_STRUC	*STRUC*		*DESC_CACHE*	*STRUC*	
REG_VAL	*DW*	?		*DB*	0
	DW	0	*_Type*	*DB*	?
ENDS				*DB*	0
				DB	0
			_Addr	*DD*	?
			_Limit	*DD*	?
			ENDS		

Table 9.3 *80386* `LOADALL` ***data structure format.***

10 *DWORDs* exactly *100h* bytes beyond the beginning of the table (*ES:EDI+100h*). This data is *not* used to load any of the registers *LOADALL* does not load (*CR2*, *CR3*, *DR0-DR3*, *TR6*, *TR7*), or the Numeric Processor eXtension (NPX). At this time, the purpose of reading this data and its destination is a mystery. Figure 9.1 shows an ICE trace showing all the bus cycles associated with *LOADALL*'s execution.

As with the 286 *LOADALL*, all CPU register entries in the *LOADALL* table are in the standard Intel format where the least significant byte is at the lowest memory address. The 386 descriptor cache entries have the format shown in Table 9.4a and Table 9.4b.

Listing 9.2 shows how to test 386 *LOADALL*. This test is more comprehensive than the 286 *LOADALL* test because of the expanded capabilities of the 386 microprocessor. This test puts the CPU into various states that are illegal and are impossible to duplicate through any other software means.

LOADALL *Emulation*

Due to the large number of systems programs that use 286 *LOADALL*, all 386 and 486 BIOSes must emulate the 286 *LOADALL* instruction (opcode *0F05*). On the 386 and 486, the 286 *LOADALL* instruction generates an invalid opcode exception. The BIOS traps this exception and does its best

Offset	Description
0-3	Access Rights. The access rights dword consumes 11 bits of this 32-bit field. See Figure 9.2b for a complete description of this field.
4-7	32-bit base address of the segment in memory
8-11	32-bit segment limit.

Table 9.4a ***80386 descriptor cache entries.***

Offset	Description
0-3	Should be 0.
4-7	32-bit base address of GDTR or IDTR.
8-11	32-bit limit of GDTR or IDTR.

Table 9.4b ***GDT and IDT descriptor cache entry formats.***

to emulate the functionality of the `LOADALL` instruction, but perfect emulation is impossible without using `LOADALL` itself. Using 386 `LOADALL` to emulate 286 `LOADALL` can be done, but it has its risks. First of all, the 486 does not have a `LOADALL` instruction. Second, Intel has threatened to remove `LOADALL` from the 386 mask.

Perfect emulation is possible on the 386 by using 386 `LOADALL` to emulate 286 `LOADALL`. Listing 9.3 shows a TSR program that uses 386 `LOADALL` to emulate 286 `LOADALL`. The program first tests that you are working on a 386 before installing itself. By using this emulation program, you can guarantee perfect 286 `LOADALL` emulation.

Conclusion

`LOADALL` is a very powerful instruction, but the features that make it so powerful also make it risky. For example, `LOADALL` can put the processor in states that are otherwise impossible to duplicate through any other software means. Using `LOADALL` requires a thorough understanding of how the CPU processes register loads, the ramifications of those register loads, and careful planning. The illegally induced processor states can easily cause system crashes if not properly planned for. The best way to avoid system crashes is to avoid using `LOADALL` unless you are totally confident in your understanding of the CPU and in your programming skills.

The 286 `LOADALL` is described in a 15-page Intel-confidential document. The document describes in detail how to use the instruction and also describes many of its possible uses. `LOADALL` can be used to access extended memory while in real mode and to emulate real mode while in protected mode. Programs such as `RAMDRIVE`, `ABOVEDISC`, and OS/2 use `LOADALL`. DOS v3.3 has provisions for using `LOADALL` by leaving a 102-byte "hole" at 80:0. If you are a systems programmer and have a need to know this information, Intel will provide it, along with source code to emulate 286 `LOADALL` on the 386 (without using 386 `LOADALL`).

Unlike the 286 `LOADALL`, the 386 `LOADALL` is still an Intel top secret. I do not know of any document that describes its use or format or acknowledges its existence. Very few people at Intel will acknowledge that `LOADALL` even exists in the 80386 mask. The official Intel line is that, due to U.S. Military pressure, `LOADALL` was removed from the 80386 mask over a year ago. However, running the program in Listing 9.2 demonstrates that `LOADALL` is alive, well, and still available on the latest shipping of the 80386.

FRAME	The FRAME number is like a clock count for the CPU. At every CPU clock, the ICE takes a picture. When a valid cycle cocurs, the ICE records its occurence. Therefore, it is possible to determine how many CPU clocks a sequence of instructions takes to execute by reading this information.
TYPE	Cycle type. Shown here are F=Fetch, R=Read, and X=eXecute.
ADDRESS	The 32-bit address asserted on the CPU address bus during each cycle.
DATA	The data asserted on the CPU data bus during each cycle.
BE3#	Byte Enable pins on the CPU. These pins determine which bytes of the 32 bits of data are valid. These pins are active low, so 8 bits of data are valid for each '0'.
BE2#	
BE1#	
BE0#	
W/R#	*Write/Read. Write =1,Read=0*
D/C#	*Data/Code. Data =1,Code=0*
M/IO#	*Memory/IO. Memory=1,IO =0*

					WDM	
FRAME				*BBBB*	*/ / /*	
		ADDRESS	*DATA*	*EEEE*	*RCI*	
	TYPE			3210	0	
DEC		*HEX*	*HEX*	####	###	
5	F	0000DE40	B490070F	0000	001	*LOADALL* fetched
8	X	executed	2 bytes at DE40L			*LOADALL* begins execution
11	R	0000D8F0	01010101	0000	001	The 10 "mystery" reads, exactly 100h bytes beyond the beginning of the *LOADALL* table.
13	R	0000D8F4	02020202	0000	001	
15	R	0000D8F8	03030303	0000	001	
17	R	0000D8FC	04040404	0000	001	
19	R	0000D900	05050505	0000	001	
21	R	0000D904	06060606	0000	001	
23	R	0000D908	07070707	0000	001	
25	R	0000D90C	08080808	0000	001	
27	R	0000D910	09090909	0000	001	
29	R	0000D914	0A0A0A0A	0000	001	
31	R	0000D7F0	7FFFFFE0	0000	001	CRO
33	R	0000D7F4	00000002	0000	001	EFLAGS
35	R	0000D7F8	00000133	0000	001	EIP
37	R	0000D7FC	66666666	0000	001	EDI
39	R	0000D800	77777777	0000	001	ESI
41	R	0000D804	55555555	0000	001	EBP
43	R	0000D808	88888888	0000	001	ESP
45	R	0000D80C	22222222	0000	001	EBX
47	R	0000D810	44444444	0000	001	EDX
49	R	0000D814	33333333	0000	001	ECX
51	R	0000D818	11111111	0000	001	EAX

Figure 9.1 *ICE trace of 386* LOADALL

53	R	0000D81C	FFFF0FF0	0000	001	DR6
55	R	0000D820	0000D402	0000	001	DR7
57	R	0000D824	xxxx0000	1100	001	TR Register
59	R	0000D828	xxxx0000	1100	001	LDT Register
61	R	0000D82C	xxxx5555	1100	001	GS Register
63	R	0000D830	xxxx4444	1100	001	FS Register
65	R	0000D834	xxxx2222	1100	001	DS Register
67	R	0000D838	xxxx6666	1100	001	SS Register
69	R	0000D83C	xxxx1111	1100	001	CS Register
71	R	0000D840	xxxx3333	1100	001	ES Register
73	R	0000D844	00008900	0000	001	TSS Descriptor Cache
75	R	0000D848	00070000	0000	001	
77	R	0000D84C	00000800	0000	001	
79	R	0000D850	00000000	0000	001	IDT Descriptor Cache
81	R	0000D854	00000000	0000	001	
83	R	0000D858	000003FF	0000	001	
85	R	0000D85C	00000000	0000	001	GDT Descriptor Cache
87	R	0000D860	00000000	0000	001	
89	R	0000D864	00000000	0000	001	
91	R	0000D868	00008200	0000	001	LDT Descriptor Cache
93	R	0000D86C	00090000	0000	001	
95	R	0000D870	00000088	0000	001	
97	R	0000D874	00008300	0000	001	GS Descriptor Cache
99	R	0000D878	00050000	0000	001	
101	R	0000D87C	0000FFFF	0000	001	
103	R	0000D880	00009300	0000	001	FS Descriptor Cache
105	R	0000D884	00040000	0000	001	
107	R	0000D888	0000FFFF	0000	001	
109	R	0000D88C	00009300	0000	001	DS Descriptor Cache
111	R	0000D890	00020000	0000	001	
113	R	0000D894	0000FFFF	0000	001	
115	R	0000D898	00009300	0000	001	SS Descriptor Cache
117	R	0000D89C	00060000	0000	001	
119	R	0000D8A0	0000FFF	0000	001	
121	R	0000D8A4	00009B00	0000	001	CS Descriptor Cache
123	R	0000D8A8	0000DD30	0000	001	
125	R	0000D8AC	0000FFFF	0000	001	
127	R	0000D8B0	00009300	0000	001	ES Descriptor Cache
129	R	0000D8B4	00030000	0000	001	
131	R	0000D8B8	00FFFFFF	0000	001	

Figure 9.1 *continued*

Descriptor Cache Registers

Whether in real or protected mode, the CPU stores the base address of each segment in hidden registers called descriptor cache registers. Each time the CPU loads a segment register, the segment base address, segment size limit, and access attributes (access rights) are loaded, or "cached," into these hidden registers. To enhance performance, the CPU makes all subsequent memory references via the descriptor cache registers instead of calculating the physical address, or looking up the base address in the descriptor table. Understanding the role of these hidden registers is paramount for exploiting highly advanced programming techniques, and for exploiting the undocumented *LOADALL* instruction. Figure 9.2a shows the descriptor cache layout for the 80286, and Figure 9.2b shows the layout for the 80386 and 80486.

At power-up, the descriptor cache registers are loaded with fixed, default values; the CPU is in real-mode, and all segments are marked as read/write data segments, including the code segment (*CS*). According to Intel, each time the CPU loads a segment register in real-mode, the base address is 16 times the segment value, while the access rights and size limit attributes are given fixed, "real-mode compatible" values. This is not true. In fact, only the *CS* descriptor cache access rights get loaded with fixed values each time the segment register is loaded — and even then only when a far jump is encountered. Loading any other segment register in real mode does not change the access rights or the segment size limit attributes stored in the descriptor cache registers. For these segments, the access rights and segment size limit attributes are honored from any previous setting (see Figure 9.3). Thus it is possible to have a 4Gb,

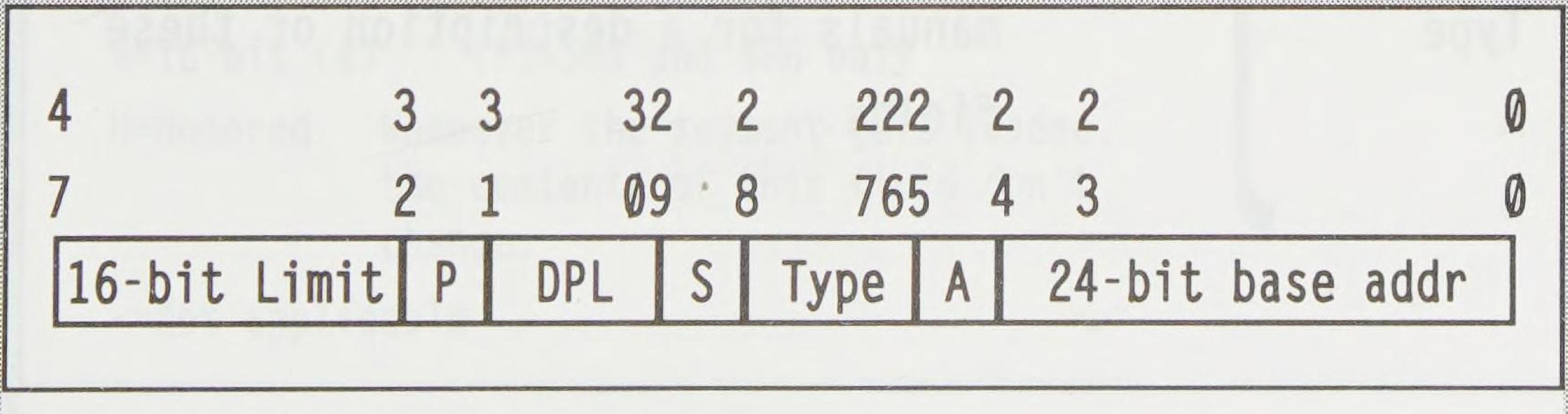


Figure 9.2a *80286 descriptor cache register.*

read-only data segment in real mode on the 80386, but Intel will not acknowledge, or support this mode of operation.

Protected mode differs from real mode in this respect: each time the CPU loads a segment register, it fully loads the descriptor cache register; no previous values are honored. The CPU loads the descriptor cache directly from the descriptor table. The CPU checks the validity of the segment by testing the access rights in the descriptor table, and illegal values will generate exceptions. Any attempt to load `CS` with a read/write data segment will

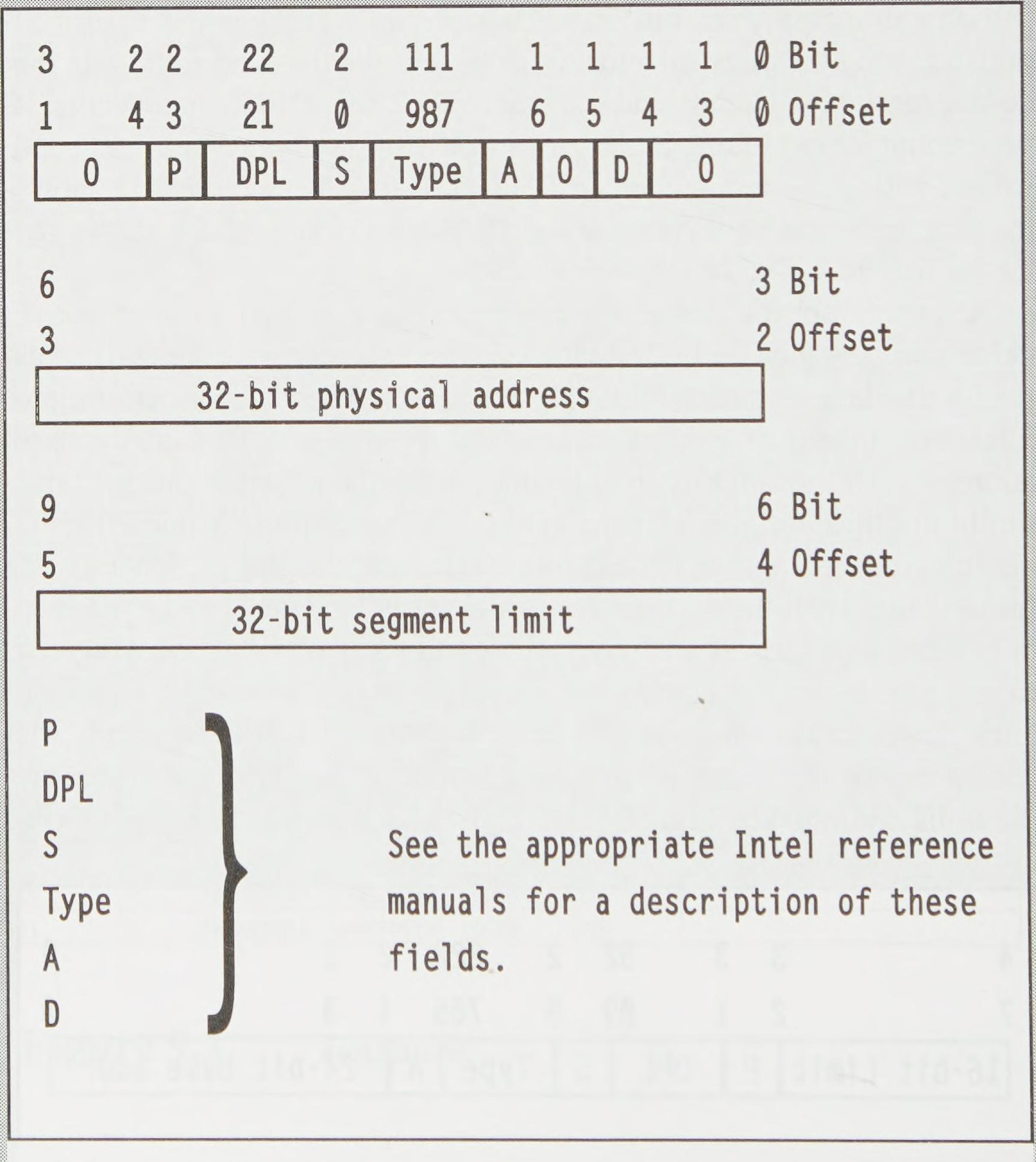
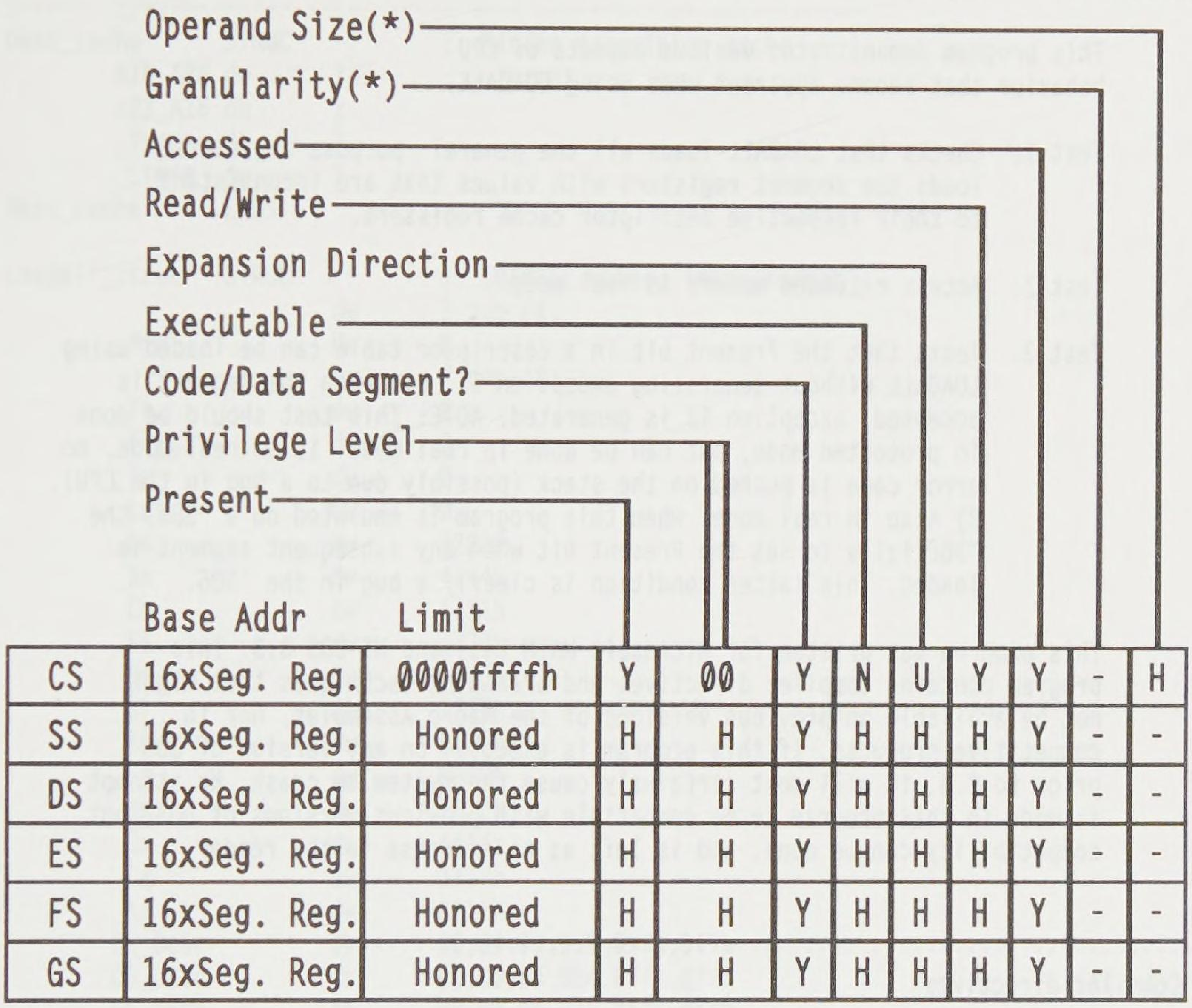

3 1
2 4
2 3
22 21
2 0
111 987
1 6
1 5
1 4
1 3
0 0
Bit
Offset
0
P
DPL
S
Type
A
0
D
0
6 3
3 2
Bit
Offset
32-bit physical address
9 5
6 4
Bit
Offset
32-bit segment limit
P
DPL
S
Type
A
D
See the appropriate Intel reference manuals for a description of these fields.

Figure 9.2b *80386/80486 descriptor cache register.*

generate a protection error. Likewise, any attempt to load a data segment register as an executable segment will also generate an exception. The CPU enforces these protection rules very strictly. If the descriptor table entry passes all the tests, then the CPU loads the descriptor cache register.


Operand Size(*)
Granularity(*)
Accessed
Read/Write
Expansion Direction
Executable
Code/Data Segment?
Privilege Level
Present
Base Addr
Limit

	Base Addr	Limit	Present	Privilege Level	Code/Data Segment?	Executable	Expansion Direction	Read/Write	Accessed	Granularity(*)	Operand Size(*)
CS	16xSeg. Reg.	0000ffffh	Y	00	Y	N	U	Y	Y	-	H
SS	16xSeg. Reg.	Honored	H	H	Y	H	H	H	Y	-	-
DS	16xSeg. Reg.	Honored	H	H	Y	H	H	H	Y	-	-
ES	16xSeg. Reg.	Honored	H	H	Y	H	H	H	Y	-	-
FS	16xSeg. Reg.	Honored	H	H	Y	H	H	H	Y	-	-
GS	16xSeg. Reg.	Honored	H	H	Y	H	H	H	Y	-	-

```
Y=Yes     (1)
N=No      (0)
U=Up      (0)
B=Byte    (0)    (*)=386 and 486 only
S=16-bit (0)    (*)=386 and 486 only
H=Honored.  Whenever the segment gets loaded,
            the contents of this field don't
            change.
-=Not applicable
```

Figure 9.3 ***Real mode descriptor cache contents.***

Listing 9.1 286LOAD.ASM

```
        Page    60,132
; ---------------------------------------------------------------
; 286LOAD.ASM   Copyright (c) 1991 Robert Collins
;
;       This program demonstrates various aspects of CPU
;       behavior that become apparent when using LOADALL.
;
;       Test 1:  Checks that LOADALL loads all the general- purpose registers;
;                loads the segment registers with values that are inconsistant
;                to their respective descriptor cache registers.
;
;       Test 2:  Access extended memory in real mode.
;
;       Test 3:  Tests that the Present bit in a descriptor table can be loaded using
;                LOADALL without generating exception 11. But when the segment is
;                accessed, exception 13 is generated. NOTE: This test should be done
;                in protected mode, but can be done in real mode. 1) In real mode, no
;                error code is pushed on the stack (possibly due to a bug in the CPU).
;                2) Also in real mode, when this program is emulated on a '386, the
;                '386 fails to set the Present bit when any subsequent segment is
;                loaded. This latter condition is clearly a bug in the '386.
;
;       This program was written for Microsoft MASM 5.1, and MS DOS 3.3. This
;       program contains compiler directives and branching techniques that might
;       not be available on previous versions of the Macro Assembler, nor in
;       competitive products. If this program is executed on any version of DOS
;       prior to 3.3, it will most certainaly cause the system to crash. No attempt
;       is made in this program to be compatible with previous versions of DOS, but
;       compatibility can be done, and is left as an exercise to the reader.
;---------------------------------------------------------------


;---------------------------------------------------------------
; Compiler directives
;---------------------------------------------------------------
        Title   LOADALL_286
        .radix  16
        .8086

;---------------------------------------------------------------
; Interrupt vector segment
;---------------------------------------------------------------
ABS0    segment at 0
        org 06h*4                       ; INT 06h vector
        INT_6           dd      ?

        org     0467h                   ; PM Return address
    PM_Ret_off      dw      ?       ; Offset
        PM_Ret_seg      dw      ?       ; Segment

        org     800h                    ; LOADALL table loc'n.
        Loadall_Locn    label   word

ABS0    ends
```

Listing 9.1 *continued*

```
;-----------------------------------------------------------------
; Structure definitions
;-----------------------------------------------------------------
Desc_cache      STRUC               ;; Hidden descriptor cache
        A15_A00 dw      ?           ;; format.
        A23_A16 db      ?
        _Type   db      ?
        _Limit  dw      ?
Desc_cache      ENDS

Loadall_struc   STRUC               ;; LOADALL memory image format
                        dw      3 dup (0)
        _Msw            dw      0
                        dw      7 dup (0)
        _Tr             dw      0
        _Flags          dw      2
        _Ip             dw      0
        _Ldt            dw      0
        _Ds             dw      2222h
        _Ss             dw      4444h
        _Cs             dw      1111h
        _Es             dw      3333h
        _Di             dw      6666h
        _Si             dw      7777h
        _Bp             dw      5555h
        _Sp             dw      8888h
        _Bx             dw      2222h
        _Dx             dw      4444h
        _Cx             dw      3333h
        _Ax             dw      1111h
        ES_Desc         db      00,00,03,93h,0ffh,0ffh
        CS_Desc         db      00,00,00,9bh,0ffh,0ffh
        SS_Desc         db      00,00,04,93h,0ffh,0ffh
        DS_Desc         db      00,00,02,93h,0ffh,0ffh
        Gdt_Desc        db      00,00,00,00h,000h,000h
        Ldt_Desc        db      00,00,06,82h,088h,000h
        Idt_Desc        db      00,00,00,00h,0ffh,003h
        TSS_Desc        db      00,00,05,89h,000h,008h
Loadall_Struc   ENDS

Descriptor  STRUC
    Seg_limit       dw      ?   ; Segment limit
    Base_A15_A00    dw      ?   ; A00..A15 of base address
    Base_A23_A16    db      ?   ; A16..A23 of base address
    Access_rights   db      ?   ; Segment access rights
    Limit_A19_A16   db      ?   ; Granularity, Op-size, Limit A16..A19
    Base_A31_A24    db      ?   ; A24..A31 of base address
Descriptor  ENDS
```

Listing 9.1 *continued*

```
INT_Desc    STRUC
    IGate_Offset    dw      ?   ; Offset of handler
    CSEG_Sel        dw      ?   ; Code segment selector
                    db      0
                    db      86h ; 286 interrupt gate=16bit CS:IP, FLAGS
    Resvd           dw      0   ; Reserved=0
INT_Desc    ENDS

;----------------------------------------------------------------
; Macro definitions
;----------------------------------------------------------------
FARJMP  MACRO   destination,selector    ; dynamic JMP FAR SEG:OFF
        db      0eah                    ;; jmp instruction
        dw      offset destination      ;; offset word
        dw      selector                ;; segment selector word
        endm

IO_DELAY        MACRO
        out     0edh,ax
        endm

LOADALL         MACRO
        mov     cx,ABS0
        mov     es,cx
        mov     cx,(size Loadall_struc) / 2
        mov     si,offset Loadall_tbl
        mov     di,800h
        rep     movsw
        db      0fh,05
        ENDM

PRINT_STRING    MACRO   MSG_NAME
        mov     ah,9
        mov     dx,offset MSG_NAME
        int     21h
        ENDM
;----------------------------------------------------------------

        _DATA    SEGMENT PARA PUBLIC 'DATA'
;----------------------------------------------------------------
; Equates & local variables
;----------------------------------------------------------------
; Protected mode access rights
;----------------------------------------------------------------
        CS_access       equ     10011011b
        DS_access       equ     10010011b
```

Listing 9.1 *continued*

```
;------------------------------------------------------------
; Text equates
;------------------------------------------------------------
        CRLF            equ     <0dh,0ah>
        CRLF$           equ     <CRLF,’$’>
        INT6            equ     [bp-4]

;------------------------------------------------------------
; Conditional compilation.  Set USE_386=1 if you plan to execute
; this program on a ’386 using EMULOAD.
;------------------------------------------------------------
        USE_386         equ     0

;------------------------------------------------------------
; Loadall table(s)
;------------------------------------------------------------
        Loadall_tbl     Loadall_struc <>
        Machine_State   Loadall_struc <>

;------------------------------------------------------------
; Global Descriptor Table
;------------------------------------------------------------
        GDT_286 Descriptor      <Gdt2_len-1,,,DS_access>
        CSEG2   Descriptor      <0ffffh,,,CS_access>  ; CS
        DSEG2   Descriptor      <0ffffh,,,DS_access>  ; DS
        Gdt2_len        equ     $-Gdt_286

;------------------------------------------------------------
; Interrupt Descriptor Table
;------------------------------------------------------------
IDT_286 INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT00
        INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT01
        INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT02
        INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT03
        INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT04
        INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT05
        INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT06
        INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT07
        INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT08
        INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT09
        INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT0a
        INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT0b
        INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT0c
        INT_Desc        <Offset INT13,CSEG2-GDT_286>  ; INT0d
IDT2_Len        equ     $-IDT_286
```

Listing 9.1 *continued*

```
;----------------------------------------------------------------
; Misc. local variables
;----------------------------------------------------------------
Mem_buffer      db      400h dup (0)
Results         dw      0
i8259_1         db      ?       ; Status for master device
i8259_2         db      ?       ; Status of slave device

;----------------------------------------------------------------
; String Messages
;----------------------------------------------------------------
Passed  db      "    PASSED.",CRLF$
Failed  db      "--> FAILED <--",CRLF$
Not_286 db      "Not 80286 class computer.",CRLF$
Rmvd    db      "LOADALL removed from 80286 mask.",CRLF$
RFail   db      "Registers weren't loaded correctly."
LF      db       CRLF$

;----------------------------------------------------------------
; I'm doing this weird string definition technique to limit the
; page width to 64 characters.
;----------------------------------------------------------------
Test_1  label   word
db      "Test 1:  Testing 286 LOADALL instruction:         ",24

Test_2  label   word
db      "Test 2:  Testing extended memory in real mode:    ",24

Test_3  label word
db      "Test 3:  Testing Present BIT in descriptor:       ",24

_DATA    ends

        _TEXT   SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:_TEXT, DS:_DATA, ES:_DATA, SS:STACK
        .286p
;----------------------------------------------------------------
; A little CS-relative data for the stack pointer. This is
; to avoid using other kludge techniques, caused by using
; LOADALL, that make using the data segment undesirable.
;----------------------------------------------------------------
        Stack_ptr       dw      0
                        dw      0
```

Listing 9.1 *continued*

```
;------------------------------------------------------------------
 LOADALL_286   proc    far
;------------------------------------------------------------------
       PUSH    DS                       ; Setup the stack to
       XOR     AX,AX                    ;  return to DOS
       PUSH    AX

       MOV     AX,_Data
       MOV     DS,AX
       MOV     ES,AX

;------------------------------------------------------------------
; Check CPU type, and set up a minimal invalid opcode handler
; in case LOADALL has been removed from the CPU mask.
;------------------------------------------------------------------
      IFE      USE_386
       Call    CPU_Type                 ; 286, 386?
       cmp     ax,2                     ; 286?
       je      short @F                 ; yep
       Print_String    LF
       Print_String    Not_286
       retf                             ; go split

@@:    enter   4,0                      ; create stack frame
       mov     word ptr INT6,offset INT6_handler
       mov     INT6[2],cs
       call    set_INT6_vector          ; set our INT6 handler
      ENDIF

       cli
       Call    Save_State               ; Save the current CPU
       Print_String    LF               ;  state
       Print_String    Test_1
```

Listing 9.1 *continued*

```
;---------------------------------------------------------------
;
; TEST1:  Real mode
;         Test general purpose registers
;         Test Segment registers
;         Test Descriptor cache base address
;
;  (1) Setup LOADALL structures, and pointers
;  (2) Execute LOADALL
;  (3) Verify results of the test
;
;---------------------------------------------------------------
        mov     ax,cs                   ; Prepare 24-bit
        mov     es,ax                   ;  physical address that
        mov     si,0                    ;  is put in the LOADALL
        call    Calc_pm_address         ;  descriptor cache
        mov     Loadall_tbl.CS_Desc.A15_A00,ax  ; entry.
        mov     Loadall_tbl.CS_Desc.A23_A16,dl
        smsw    ax
        mov     Loadall_tbl._Msw,ax
        mov     Loadall_tbl._Ip,offset Verify_State
        mov     word ptr cs:stack_ptr,sp       ; save SS:SP
        mov     word ptr cs:stack_ptr[2],ss
        LOADALL                         ; If LOADALL is removed
        nop                             ;  from the CPU mask,
        Print_String    failed          ;  then fall through
        Print_String    Rmvd            ;  to here.

Loadall_RET:
        call    Restore_state

      IFE       USE_386
        call    set_INT6_vector         ; set our INT6 handler
        leave
      ENDIF

        retf
```

Listing 9.1 *continued*

```
;----------------------------------------------------------------
  Verify_State:                   ; Verify that LOADALL worked
;----------------------------------------------------------------
; This is where we land for the first test of '286 LOADALL. The purpose
; of this test is to verify that all the general purpose registers get
; loaded correctly. Specifically, we are testing to verify that all
; segment registers contain values that don't correspond to the memory
; addresses they appear to be pointing to. In other words, we are
; checking that the segment registers have one value, while their
; associated hidden descriptor cache registers have different values.
;----------------------------------------------------------------
        cmp     ax,1111h                ; Test AX
        jne     @F
        cmp     bx,2222h                ; Test BX
        jne     @F
        cmp     cx,3333h                ; Test CX
        jne     @F
        cmp     dx,4444h                ; Test DX
        jne     @F
    cmp     bp,5555h                    ; Test BP
        jne     @F
        cmp     di,6666h                ; Test DI
        jne     @F
        cmp     si,7777h                ; Test SI
        jne     @F
        cmp     sp,8888h                ; Test SP
        jne     short @F
        mov     ax,cs                   ; Test CS
        cmp     ax,1111h
        jne     short @F
        mov     ax,ds                   ; Test DS
        cmp     ax,2222h
        jne     short @F
        mov     ax,es                   ; Test ES
        cmp     ax,3333h
        jne     short @F
        mov     ax,ss                   ; Test SS
        cmp     ax,4444h
        jne     short @F
        cmp     word ptr ds:[0],0202h   ; Test DS Desc Cache
        jne     short @F
        cmp     word ptr es:[0],0303h   ; Test ES Desc Cache
        jne     short @F
        cmp     word ptr ss:[0],0404h   ; Test SS Desc Cache
        jne     short @F
        mov     ax,_Data
        mov     ds,ax
        mov     es,ax

        mov     ax,cs:stack_ptr[2]
        mov     ss,ax
        mov     sp,cs:stack_ptr
        FARJMP  <@Test1_Pass>,<seg _Text>
```

Listing 9.1 *continued*

```
;-----------------------------------------------------------------
; Loadall failed the REGISTERs test.
;-----------------------------------------------------------------
@@:     mov     ax,_Data
        mov     ds,ax
        mov     es,ax
        mov     ax,cs:stack_ptr[2]
        mov     ss,ax
        mov     sp,cs:stack_ptr
        FARJMP  <@F>,<seg _Text>

@@:     Print_String    failed
        Print_String    RFail
        jmp     loadall_ret

;-----------------------------------------------------------------
; LOADALL passed
;-----------------------------------------------------------------
@Test1_Pass:
        Print_String    passed

;-----------------------------------------------------------------
;
; TEST2:  Access extended memory while in real mode.
;
;  (1) Enable A20
;  (2) Save contents of extended memory
;  (3) Write data pattern in extended memory
;  (4) Set IP & ES descriptor cache pointing to extended memory
;  (5) LOADALL
;  (6) Verify results
;  (7) Restore original data in extended memory
;
;-----------------------------------------------------------------
```

Listing 9.1 *continued*

```
        Print_String    Test_2
        Call    Enable_Gate20           ; Enable extended memory
        mov     bx,0ffffh               ; Point to extended mem.
        mov     ds,bx                   ; as FFFF:0010
        mov     si,10h
        mov     di,offset Mem_buffer
        mov     cx,400h / 2             ; 1k data block to test
        rep     movsw                   ; save extended memory
        mov     ax,5aa5h                ; test pattern
        mov     es,bx                   ; point to extended mem.
        mov     cx,400h / 2
        mov     di,10h
        rep     stosw                   ; store pattern in mem.
        mov     ax,_data
        mov     ds,ax
        mov     Loadall_tbl._AX,5aa5h
        mov     Loadall_tbl._CX,400h / 2
        mov     Loadall_tbl._DI,0h
        mov     Loadall_tbl._SP,sp      ; save SP
        mov     Loadall_tbl._IP,offset @F
        mov     Loadall_tbl.ES_Desc.A15_A00,00
        mov     Loadall_tbl.ES_Desc.A23_A16,10
        LOADALL
@@:     repz    scasw                   ; data match?
        lahf                            ; get flags
        mov     bx,_Data
        mov     ds,bx
        mov     cx,0ffffh
        mov     es,cx
        mov     cx,400h / 2
        mov     si,offset Mem_buffer
        mov     di,10h
        rep     movsw                   ; restore data
        mov     es,bx
        mov     cx,cs:stack_ptr[2]
        mov     ss,cx
        FARJMP  <@F>,<seg _Text>
@@:     sahf                            ; restore flags
        jz      @Test2_Pass
        PRINT_STRING    failed
        jmp     Loadall_RET

@Test2_Pass:
        PRINT_STRING    passed
```

Listing 9.1 *continued*

```
;-------------------------------------------------------------
;
; TEST3:  Test that the Present bit gets loaded w/out exception,
;         but when a segment is accessed INT13 get generated.
;
;      If LOADALL works even remotely like we think it does,
;         then this test will work in REAL MODE!  And as this
;         test was originally programmed, it did!  However, I
;         was doing my testing by emulating '286 LOADALL with
;         '386 LOADALL, where I could use an ICE for debug
;         purposes.  The code worked on a '286, but failed on a
;         '386!  I found that the '386 fails to clear the
;         Present bit in the descriptor cache register when a
;         segment register is loaded in real mode.  This is
;         obviously a bug in the '386.  Since the CPU is in a
;         state that can never be duplicated under any program
;         control, except by using LOADALL (Present=0), the bug
;         will never be manifested in any production code.  As a
;         result, I reprogrammed this example to use protected
;         mode so it would work on both the '286 and '386.
;
;  (1) Prepare GDT & IDT descriptor cache registers and
;      descriptor tables, & segment selectors
;  (2) Set protected mode bit, clear Present bit, set IP
;  (3) Save the 8259 masks, set the PM return address, and set
;      CMOS shutdown=5
;  (4) LOADALL
;  (5) Generate the exception
;  (6) Reset ES to a valid segment selector & save results of
;      test.
;  (7) Reset the CPU, restore 8259 masks, inhibit A20 from the
;      CPU bus, restore segment registers to real mode values.
;  (8) Verify the results
;
;-------------------------------------------------------------
; Test Present bit:  verify that P=0 in a descriptor cache
; register will, even in REAL MODE, will generate an exception
; 13 when trying to access memory
;-------------------------------------------------------------
```

Listing 9.1 *continued*

```
Test3:  Print_String    Test_3
        mov     ax,_Data
        mov     es,ax
        mov     si,0
        call    Calc_pm_address
        mov     DSEG2.Base_A15_A00,ax
        mov     DSEG2.Base_A23_A16,dl
        add     ax,offset GDT_286
        adc     dl,0
        mov     Loadall_tbl.GDT_Desc.A15_A00,ax
        mov     Loadall_tbl.GDT_Desc.A23_A16,dl
        mov     Loadall_tbl.GDT_Desc._Limit,GDT2_Len-1
        mov     si,offset IDT_286
        call    Calc_pm_address
        mov     Loadall_tbl.IDT_Desc.A15_A00,ax
        mov     Loadall_tbl.IDT_Desc.A23_A16,dl
        mov     Loadall_tbl.IDT_Desc._Limit,IDT2_Len-1
        mov     ax,_TEXT
        mov     es,ax
        mov     si,0
        call    Calc_pm_address
        mov     CSEG2.Base_A15_A00,ax
        mov     CSEG2.Base_A23_A16,dl
        mov     Loadall_tbl._CS,CSEG2-GDT_286

        or      Loadall_tbl._MSW,1
    and     Loadall_tbl.ES_Desc._Type,7fh   ; Clear P bit
        mov     Loadall_tbl._IP,offset @PM_286  ; Set IP

        Call    Get_INT_Status          ; save PIC masks
        mov     ax,offset @RM_286       ; save real mode return
        Call    SetPM_RET_addr          ;  address
        Call    Set_shutdown_type       ; set shutdown in CMOS

        LOADALL
```

Listing 9.1 *continued*

```
@PM_286:mov     al,es:[di][2]

        mov     ax,DSEG2-GDT_286
        mov     es,ax
        mov     ES:Results,di
        jmp     RESET_CPU

@RM_286:mov     ax,cs:stack_ptr[2]
        mov     ss,ax
        mov     sp,cs:stack_ptr
        mov     ax,_Data
        mov     ds,ax
        mov     es,ax
        call    Set_INT_Status
        call    Shut_A20

        mov     di,Results              ; If an exception 13 was
        test    di,1                    ;  generated, then the
        jnz     @F                      ;  low bit of DI is set

        Print_String    failed          ; Test failed
        jmp     Loadall_RET

@@:     Print_String    passed          ; Test passed
        jmp     Loadall_RET
LOADALL_286     endp
```

Listing 9.1 *continued*

```
;-------------------------------------------------------------------
; Minimal exception 13 handler that points past a 4-byte opcode,
; and sets the lowest bit in DI before returning.
;-------------------------------------------------------------------
       INT13    label   word
       push     bp
       mov      bp,sp
       add      word ptr [bp][4],4
       or       di,1
       pop      bp
       add      sp,2
       iret

;-------------------------------------------------------------------
; Equates & local variables
;-------------------------------------------------------------------
; I/O Ports
;-------------------------------------------------------------------
       Mstrmsk          equ     021h    ; 8259 master mask addr
       KBC_CTL          equ     060h    ; 8042 control port
   KBC_STAT         equ     064h    ; 8042 status port
       Cmos_index       equ     070h    ; CMOS address port
       Cmos_data        equ     071h    ; CMOS data port
       Slv_msk          equ     0a1h    ; 8259 slave mask addr

;-------------------------------------------------------------------
; CMOS RAM
;-------------------------------------------------------------------
       Shut_down        equ     00fh    ; CMOS index for shutdwn
       Type5            equ     5       ; Shutdown type-5

;-------------------------------------------------------------------
; Keyboard Controller
;-------------------------------------------------------------------
       inpt_buf_full    equ     2       ; Input buffer full
       Shutdown_CMD     equ     0feh    ; Shutdown CMD for KBC
       enable_bit20     equ     0dfh    ; enable A20 command
       disable_bit20    equ     0ddh    ; disable A20 command

;-------------------------------------------------------------------;
  RESET_CPU:; Resets the CPU by sending a shutdown command to the keyboard controller.
;-------------------------------------------------------------------
; Input:   None
; Output:  None
; Register(s) modified:  Doesn't matter, the CPU is reset
;-------------------------------------------------------------------
       mov      al,Shutdown_CMD ; get shutdown command
       out      KBC_STAT,al     ; send command to shutdown CPU
       cli                      ; disable interrupts so thaT an INT can't
                                ;  come through before the CPU resets
       hlt                      ;
```

Listing 9.1 *continued*

```
;-------------------------------------------------------------;
; SETPM_RET_ADDR:  Save the real-mode return address @ 40:67 from protected mode.
;---------------------------------------------------------------
; Input:   CS:AX = Return address from PM.
;          DS    = Better darn well have a PM segment selector! (Or else Kablooie!)
; Output:  None
; Register(s) modified:  None
;---------------------------------------------------------------
  Setpm_ret_addr        proc    near
;---------------------------------------------------------------
        push    dx              ; save it
        push    ds
        mov     dx,ABS0         ;
        mov     ds,dx
ASSUME  DS:ABS0
        mov     DS:PM_Ret_off,ax
        mov     DS:PM_Ret_seg,cs
ASSUME  DS:_DATA
        pop     ds
        pop     dx
        ret
Setpm_ret_addr  endp

;---------------------------------------------------------------
; Get_INT_status:  Saves the master and slave mask register
;                  contents from the 8259 interrupt controller.
;---------------------------------------------------------------
; Input:   DS      = _DATA SEGMENT
; Output:  i8259_1 = Status of master device
;          i8259_2 = Status of slave device
; Register(s) modified:  None
;---------------------------------------------------------------
  Get_int_status        proc    near
;---------------------------------------------------------------
        push    ax
        in      al,mstrmsk      ; get master PIC mask
        mov     i8259_1,al
       IO_Delay                 ; I/O delay
        in      al,slv_msk      ; get slave PIC mask
        mov     i8259_2,al
        pop     ax
        ret                     ; exit
Get_int_status  endp
```

Listing 9.1 *continued*

```
;--------------------------------------------------------------;
; Set_INT_status:  Restores the interrupt status of the 8259A
;                  programmable interrupt controller (PIC).
;---------------------------------------------------------------
; Input:   i8259_1 = Status of master device
;          i8259_2 = Status of slave device
;          DS      = _DATA SEGMENT
; Output:  None
; Register(s) modified:  None
;---------------------------------------------------------------
  Set_int_status        proc    near
;---------------------------------------------------------------
        pushf                   ; save interrupt flag
        cli                     ; we REALLY don't want an int to come through
        push    ax              ;  while we are reprogramming the PIC masks
        mov     al,i8259_1
        out     mstrmsk,al      ; restore master PIC mask
        IO_Delay                ; I/O delay
        mov     al,i8259_2
        out     slv_msk,al      ; restore slave PIC mask
        pop     ax
        popf                    ; restore interrupt flag
        ret                     ; exit
Set_int_status  endp

;---------------------------------------------------------------
; SET_SHUTDOWN_TYPE:  Set the processor shutdown type-5 in CMOS.
;---------------------------------------------------------------
; Input:   None
; Output:  None
; Register(s) modified:  None
;---------------------------------------------------------------
  Set_shutdown_type     proc    near
;---------------------------------------------------------------
    pushf                   ; save interrupt status
        cli                     ; disable ints so somebody else doesn't do this right now
        push    ax
        mov     al,shut_down    ; Set shutdown byte
        out     cmos_index,al   ; to shut down x05.
        IO_Delay                ; I/O delay
        mov     al,Type5        ;
        out     cmos_data,al    ; CMOS data port
        pop     ax
        popf
        ret
set_shutdown_type       endp
```

Listing 9.1 *continued*

```
;----------------------------------------------------------------
; Enable_gate20:  Turn on A20, and check for errors.
;----------------------------------------------------------------
; Input:   None
; Output:  CY=ERROR
; Register(s) modified:  None
;----------------------------------------------------------------
  Enable_gate20 proc    near
;----------------------------------------------------------------
        push    ax
        mov     ah,enable_bit20          ; gate address bit 20 on
        Call    Gate_A20
        or      al,al                    ; command accepted?
        jz      A20_OK                   ; go if yes
        stc                              ; set error flag
A20_OK: pop     ax
        ret                              ; exit
Enable_gate20   endp

;----------------------------------------------------------------
; SHUT_A20:  Disable A20 from CPU address BUS.
;----------------------------------------------------------------
; Input:   None
; Output:  CY=ERROR
; Register(s) modified:  None
;----------------------------------------------------------------
  Shut_a20      proc    near
;----------------------------------------------------------------
        push    ax
        mov     ah,disable_bit20         ; gate address bit 20 on
        Call    Gate_A20
        or      al,al                    ; was command accepted?
        jz      A20_Shut                 ; go if yes
        stc                              ; set error flag

A20_Shut:
        pop     ax
        ret                              ; exit
Shut_a20        endp
```

Listing 9.1 *continued*

```
;----------------------------------------------------------------
; GATE_A20:  This routine controls a signal which gates address
;            line 20 (A20).  The gate A20 signal is an output of
;        of the 8042 slave processor (keyboard controller).
;            A20 should be gated on before entering protected
;            mode, to allow addressing of the entire 16M address
;            space of the 80286, or 4G address space of the
;            80386 & 80486.  It should be gated off after
;            entering real mode -- from protected mode.
;----------------------------------------------------------------
; Input:   AH = DD ==> A20 gated off (A20 always 0)
;          AH = DF ==> A20 gated on  (CPU controls A20)
; Output:  AL = 0  ==> Operation successful
;          AL = 2  ==> Operation failed, 8042 can't accept cmd
; Register(s) modified:  AX
;----------------------------------------------------------------
  Gate_a20      proc    near
;----------------------------------------------------------------
        pushf                       ; save interrupt status
        cli                         ; disable ints while using 8042
        Call    Empty_8042          ; insure 8042 input buffer empty
        jnz     A20_Fail            ; ret: 8042 unable to accept cmd
        IO_Delay                    ; I/O Delay
        mov     al,0D1h             ; 8042 cmd to write output port
        out     KBC_STAT,al         ; output cmd to 8042
        Call    Empty_8042          ; wait for 8042 to accept cmd
        jnz     A20_Fail            ; ret: 8042 unable to accept cmd
        mov     al,ah               ; 8042 port data
        out     KBC_CTL,al          ; output port data to 8042
        Call    Empty_8042          ; wait for 8042 to port data
        push    cx                  ; save it
        mov     cx,14h              ;
@DLY:   IO_Delay                    ; Wait for KBC to execute the
        loop    @DLY                ;  command.  (about 25uS)
        pop     cx                  ; restore it

A20_Fail:
        popf                        ; restore flags
        ret
Gate_a20        endp
```

Listing 9.1 *continued*

```
;---------------------------------------------------------------
; EMPTY_8042:  This routine waits for the 8042 buffer to empty.
;---------------------------------------------------------------
; Input:   None
; Output:  AL = 0, 8042 input buffer empty:      ZF
;          AL = 2, Time out; 8042 buffer full:  NZ
; Register(s) modified:  AX
;---------------------------------------------------------------
  Empty_8042    proc    near
;---------------------------------------------------------------
        push    cx              ; save CX
        xor     cx,cx           ; CX=0:  timeout value

Try_KBC:
        IO_Delay                ;
        in      al,KBC_STAT     ; read 8042 status port
        and     al,inpt_buf_full; input buffer full flag (D1)
        loopnz  Try_KBC         ; loop until input buffer empty or timeout
        pop     cx              ; restore CX
        ret
Empty_8042  endp

;---------------------------------------------------------------
; CALC_PM_ADDRESS:  Calculate 32-bit protected mode address.
;                   Used for building descriptor tables.
;---------------------------------------------------------------
; Input:   ES:SI = Real mode address
; Output:  DX:AX = 32-bit linear address
; Register(s) modified:  AX, DX
;---------------------------------------------------------------
  Calc_pm_address       proc    near
;---------------------------------------------------------------
        mov     ax,es           ; point to control block
        xor     dh,dh           ; clear upper register
        mov     dl,ah           ; build high byte of 32-bit addr
        shr     dl,4            ; use only high nibble from (AX)
        shl     ax,4            ; strip high nibble from segment
        add     ax,si           ; add GDT offset for low word
        adc     dx,0            ; adj high byte if CY from low
        ret                     ; back to calling program
calc_pm_address endp
```

Listing 9.1 *continued*

```
;-----------------------------------------------------------------
  Save_state      proc     near    ; Save the machine state before LOADALL
;-----------------------------------------------------------------
         push    ax
         push    ds
         mov     si,0
         mov     di,offset Machine_State.ES_Desc
         mov     ax,3000h                    ; ES descriptor
         mov     ds,ax
         mov     bx,0303h
         movsw
         mov     word ptr [si-2],bx

         add     ax,1000h                    ; SS descriptor
         add     bx,0101h
         mov     si,0
         add     di,0ah
         mov     ds,ax
         movsw
         mov     word ptr [si-2],bx
         sub     ax,2000h                    ; DS descriptor
         sub     bx,0202h
         mov     si,0
         add     di,4
         mov     ds,ax
         movsw
         mov     word ptr [si-2],bx
         pop     ds

         smsw    ax
         mov     Machine_State._Msw,ax
         pushf
         pop     ax
         mov     Machine_State._Flags,ax
         pop     ax

    mov      Machine_State._DI,di
         mov     Machine_State._SI,si
         mov     Machine_State._BP,bp
         mov     Machine_State._BX,bx
         mov     Machine_State._DX,dx
         mov     Machine_State._CX,cx
         mov     Machine_State._AX,ax
         mov     ax,ds
         mov     Machine_State._DS,ax
         mov     ax,es
         mov     Machine_State._ES,ax
         ret
Save_state       endp
```

Listing 9.1 *continued*

```
;---------------------------------------------------------------
 Restore_state proc     near     ; Restore the machine state after LOADALL
;---------------------------------------------------------------
        mov     ax,_data
        mov     ds,ax
        mov     ax,3000h                ; ES
        mov     es,ax
        mov     si,offset Machine_State.DS_Desc
        mov     di,0
        movsw
        add     ax,1000h                ; SS
        add     si,0ah
        mov     di,0
        mov     es,ax
        movsw
        sub     ax,2000h                ; DS
        add     si,4
        mov     di,0
        mov     es,ax
        movsw
        mov     ax,Machine_State._ES
        mov     es,ax
        mov     ax,Machine_State._DS
        mov     ds,ax
        mov     ax,Machine_State._Flags
        push    ax
        popf
        mov     ax,Machine_State._Msw
        lmsw    ax
        mov     ax,Machine_State._AX
        mov     cx,Machine_State._CX
        mov     dx,Machine_State._DX
        mov     bx,Machine_State._BX
        mov     bp,Machine_State._BP
        mov     si,Machine_State._SI
        mov     di,Machine_State._DI
        ret
Restore_State   endp

      IFE       USE_386
;---------------------------------------------------------------
; Include the CPU_TYPE procedure & LOADALL test
;---------------------------------------------------------------
    Include CPU_TYPE.ASM
      ENDIF

_text   ends

        stack   segment para stack 'stack'
        db      400h dup (0)
        stack   ends
        end     LOADALL_286
; End of File
```

Listing 9.2 386LOAD.ASM

```
        Page    60,132
;-----------------------------------------------------------------
;
; 386LOAD.ASM  Copyright (c) 1991 Robert Collins
;
;       This program demonstrates various aspects of CPU
;       behavior that become apparent when using LOADALL.
;
;       Test 1:  Checks that LOADALL loads all the general-
;                purpose registers; loads the segment registers
;                with values that are inconsistant to their
;                respective descriptor cache registers.
;
;       Test 2:  Access extended memory in real mode.
;
;       Test 3:  Tests that the Present bit in a descriptor
;                table can be loaded using LOADALL without
;                generating exception 11.  But when the segment
;                is accessed, exception 13 is generated.
;                NOTE:  This test should be done in protected
;                mode, but can be done in real mode.  1) In real
;                mode, no error code is pushed on the stack
;                (possibly due to a bug in the CPU).  2) Also,
;                when this test is executed in real-mode, the
;                '386 fails to set the Present bit when the
;                segment is subsequently loaded.  This latter
;                condition is clearly a bug in the '386.
;
;       Test 4:  Test 32-bit protected mode.
;
;       Test 5:  Test 32-bit real mode -- a mode that is an illegal mode for the CPU.
;
;       Test 6:  Test that the Granularity bit in the descriptor
;                cache register has no affect on the segment limit field.
;
;       Test 7:  Test execution breakpoints.
;
;       Test 8:  Test data breakpoints.
;
;       This program was written for Microsoft MASM 5.1.
;       This program contains compiler directives and branching
;       techniques that might not be available on previous
;       versions of the Macro Assembler, nor in competitive products.
;
;-----------------------------------------------------------------


;-----------------------------------------------------------------
; Compiler directives
;-----------------------------------------------------------------
        Title   LOADALL_386
        .radix  16
        .8086
```

Listing 9.2 *continued*

```
;---------------------------------------------------------------
; Interrupt vector segment
;---------------------------------------------------------------
ABS0    segment at 0
        org 6*4
        INT_6           dd      ?
ABS0    ends

;---------------------------------------------------------------
; Structure definitions
;---------------------------------------------------------------
Desc_cache      STRUC
                db      0
        _Type   db      ?
        _CS32   db      0
                db      0
        _Addr   dd      ?
        _Limit  dd      ?
Desc_cache      ENDS

Loadall_struc   STRUC
        _Cr0            dd      0
        _Eflags         dd      2
        _Eip            dd      0
        _Edi            dd      66666666h
        _Esi            dd      77777777h
        _Ebp            dd      55555555h
        _Esp            dd      88888888h
        _Ebx            dd      22222222h
        _Edx            dd      44444444h
        _Ecx            dd      33333333h
        _Eax            dd      11111111h
        _Dr6            dd      0
        _Dr7            dd      0
        _Tr             dd      0
        _Ldt            dd      0
        _Gs             dd      5555h
        _Fs             dd      4444h
        _Ds             dd      2222h
        _Ss             dd      6666h
        _Cs             dd      1111h
        _Es             dd      3333h
        TSS_Desc        dd      00008900h,00070000h,00000800h
        IDT_Desc        dd      00000000h,00000000h,000003ffh
        Gdt_Desc        dd      00000000h,00000000h,00000000h
        Ldt_Desc        dd      00008200h,00090000h,00000088h
        GS_Desc         dd      00009300h,00050000h,0000ffffh
        FS_Desc         dd      00009300h,00040000h,0000ffffh
        DS_Desc         dd      00009300h,00020000h,0000ffffh
        SS_Desc         dd      00009300h,00060000h,0000ffffh
        CS_Desc         dd      00009b00h,00000000h,0000ffffh
        ES_Desc         dd      00009300h,00030000h,00fffffch
Loadall_Struc   ENDS
```

Listing 9.2 *continued*

```
Descriptor STRUC
    Seg_limit       dw      ?       ; Segment limit
    Base_A15_A00    dw      ?       ; A00..A15 of base address
    Base_A23_A16    db      ?       ; A16..A23 of base address
    Access_rights   db      ?       ; Segment access rights
    Limit_A19_A16   db      ?       ; Granularity, Op-size, Limit A16..A19
    Base_A31_A24    db      ?       ; A24..A31 of base address
Descriptor ENDS

INT_Desc   STRUC
    IGate_Offset    dw      ?       ; Offset of handler
    CSEG_Sel        dw      ?       ; Code segment selector
                    db      0
                    db      86h     ; 286 interrupt gate=16bit CS:IP, FLAGS
    Resvd           dw      0       ; Reserved=0
INT_Desc   ENDS

;-------------------------------------------------------------------------------
; Macro definitions
;-------------------------------------------------------------------------------
    Init_descriptor    macro    segment,offset,desc_name
        push    ax
        push    dx
        push    si
        push    es
        mov     ax,&segment                 ;; get segment name
        mov     es,ax                       ;;   to form 24 bit address
        mov     si,&offset                  ;;
        mov     ax,es                       ; point to control block
        xor     dh,dh                       ; clear upper register
        mov     dl,ah                       ; build high byte of 32-bit address
        shr     dl,4                        ; use only high nibble from (AX)
        shl     ax,4                        ; strip high nibble from segment
        add     ax,si                       ; add the GDT offset to develop low word
        adc     dx,0                        ; adjust high byte if carry from low
        mov     &desc_name.Base_A15_A00,ax        ;; low word of address
        mov     &desc_name.Base_A23_A16,dl        ;; high byte of address
        mov     &desc_name.Base_A31_A24,dh        ;; high byte of linear address
        pop     es
        pop     si
        pop     dx
        pop     ax
    endm

FARJMP  MACRO   destination,selector        ; dynamic JMP FAR SEG:OFF
        db      0eah                        ;; jmp instruction
        dw      offset destination          ;; offset word
        dw      selector                    ;; segment selector word
        endm
```

Listing 9.2 *continued*

```
LONGJMP MACRO   destination,selector    ; dynamic JMP FAR SEG:OFF
        db      0eah                    ;; jmp instruction
        dd      offset destination      ;; offset word
        dw      selector                ;; segment selector word
        endm

IO_DELAY    MACRO
        out     0edh,ax
        endm

LOADALL         MACRO
        db      0fh,07h
        ENDM

PRINT_STRING    MACRO   MSG_NAME
        mov     ah,9
        mov     dx,offset MSG_NAME
        int     21h
        ENDM

        _DATA    SEGMENT PARA PUBLIC 'DATA'
;---------------------------------------------------------------
; Equates & local variables
;---------------------------------------------------------------
; Protected mode access rights
;---------------------------------------------------------------
        CS_access       equ     10011011b
        DS_access       equ     10010011b

;---------------------------------------------------------------
; Text equates
;---------------------------------------------------------------
        CRLF            equ     <0dh,0ah>
        CRLF$           equ     <CRLF,'$'>
        INT6            equ     [bp-4]

;---------------------------------------------------------------
; Loadall table(s)
;---------------------------------------------------------------
        Loadall_tbl     Loadall_struc <>
        Machine_State   Loadall_struc <>

;---------------------------------------------------------------
; Global Descriptor Table
;---------------------------------------------------------------
        GDT_386 Descriptor      <Gdt3_len-1,,,DS_access>
        CSEG3   Descriptor      <0ffffh,0,0,CS_access>
        DSEG3   Descriptor      <0ffffh,0,20h,DS_access>
        Gdt3_len      equ       $-Gdt_386
```

Listing 9.2 *continued*

```
;---------------------------------------------------------------------
; Interrupt Descriptor Table
;---------------------------------------------------------------------
IDT_386 INT_Desc        <Offset INT13,CSEG3-GDT_386>  ; INT00
        INT_Desc        <Offset INT01,CSEG3-GDT_386>  ; INT01
        INT_Desc        <Offset INT13,CSEG3-GDT_386>  ; INT02
        INT_Desc        <Offset INT13,CSEG3-GDT_386>  ; INT03
        INT_Desc        <Offset INT13,CSEG3-GDT_386>  ; INT04
        INT_Desc        <Offset INT13,CSEG3-GDT_386>  ; INT05
        INT_Desc        <Offset INT13,CSEG3-GDT_386>  ; INT06
        INT_Desc        <Offset INT13,CSEG3-GDT_386>  ; INT07
        INT_Desc        <Offset INT13,CSEG3-GDT_386>  ; INT08
        INT_Desc        <Offset INT13,CSEG3-GDT_386>  ; INT09
        INT_Desc        <Offset INT13,CSEG3-GDT_386>  ; INT0a
        INT_Desc        <Offset INT13,CSEG3-GDT_386>  ; INT0b
        INT_Desc        <Offset INT13,CSEG3-GDT_386>  ; INT0c
        INT_Desc        <Offset INT13,CSEG3-GDT_386>  ; INT0d
IDT3_Len        equ     $-IDT_386

;---------------------------------------------------------------------
; Misc. local variables
;---------------------------------------------------------------------
        Mem_buffer      db      400h dup (0)     ; Store 2M mem.
        Buffer          db      40h dup (0)
        Buffer2         dw      10h dup (0)

        RM_IDT3_Ptr     dw      (256d*4)-1       ; Real-mode IDT
                        dd      0                ;  pointer
                        dw      0

;---------------------------------------------------------------------
; String Messages
;---------------------------------------------------------------------
Passed  db      "    PASSED.",CRLF$
Failed  db      "--> FAILED <--",CRLF$
Not_386 db      "Not 80386 class computer.",CRLF$
Rmvd    db      "LOADALL removed from 80386 mask.",CRLF$
RFail   db      "Registers weren't loaded correctly."
LF      db       CRLF$
```

Listing 9.2 *continued*

```
;----------------------------------------------------------------
; I'm doing this weird string definition technique to limit the
; page width to 64 characters.
;----------------------------------------------------------------
Test_1  label   word
db      "Test 1:  Testing 386 LOADALL instruction:         ",24

Test_2  label   word
db      "Test 2:  Testing extended memory in real mode:    ",24

Test_3  label   word
db      "Test 3:  Testing Present BIT in descriptor:       ",24

Test_4  label   word
db      "Test 4:  Testing 32-bit protected mode:           ",24

Test_5  label   word
db      "Test 5:  Testing 32-bit real mode:                ",24

Test_6  label   word
db      "Test 6:  Testing Granularity BIT:                 ",24

Test_7  label   word
db      "Test 7:  Testing Execution breakpoints:           ",24

Test_8  label   word
db      "Test 8:  Testing byte, write, data breakpoints:   ",24

_DATA   ENDS

        _TEXT   SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:_TEXT, ds:_DATA, ES:_DATA, SS:STACK
    .386P
;----------------------------------------------------------------
; A little CS-relative data for the stack pointer.  This is
; to avoid using other kludge techniques, caused by using
; LOADALL, that make using the data segment undesirable.
;----------------------------------------------------------------
        Stack_ptr       dd      0
                        dw      0

;----------------------------------------------------------------
  LOADALL_386   proc    far
;----------------------------------------------------------------
        PUSH    DS                      ; Setup the stack to
        XOR     AX,AX                   ;  return to DOS
        PUSH    AX

        MOV     AX,_Data
        MOV     DS,AX
        MOV     ES,AX
```

Listing 9.2 *continued*

```
;---------------------------------------------------------------
; Check CPU type
;---------------------------------------------------------------
        Call    CPU_Type                ; 386, 486?
        cmp     ax,3                    ; 386?
        je      short @F                ; yep
        Print_String    LF
        Print_String    Not_386
        retf                            ; go split

@@:     enter   4,0                     ; create stack frame
        mov     word ptr INT6,offset INT6_handler
        mov     INT6+2,cs
        call    set_INT6_vector         ; set our INT6 handler
        cli
        Call    Save_State
        Print_String    LF
        Print_String    Test_1
        Init_descriptor <seg Gdt_386>,<offset Gdt_386>,Gdt_386
        Init_descriptor cs,   0,CSEG3

;---------------------------------------------------------------
; TEST1:  16-bit Real mode
;         Test general purpose registers. Test Segment registers.
;         Test Descriptor cache base address.
;
;  (1) Setup LOADALL structures, and pointers. (2) Execute LOADALL.
;  (3) Verify results of the test.
;---------------------------------------------------------------
        mov     eax,cs                  ; Prepare a 32-bit physical address that
        shl     eax,4                   ;  is put in the LOADALL CS descriptor cache.
        mov     Loadall_tbl.CS_Desc._Addr,eax

        mov     eax,cr0                 ; Initialize CR0 in the
    and     eax,0fffffff0h          ;  LOADALL data table
        mov     Loadall_tbl._CR0,eax    ;  DR6, DR7, EIP
        mov     eax,dr6
        mov     ebx,dr7
        mov     Loadall_tbl._DR6,eax
        mov     Loadall_tbl._DR7,ebx
        mov     Loadall_tbl._EIP,offset Verify_State
        mov     dword ptr cs:stack_ptr,esp       ; save SS:ESP
        mov     word ptr cs:stack_ptr[4],ss
        mov     edi,offset loadall_tbl
        LOADALL
        nop                             ; If LOADALL is removed
        Print_String    failed          ;  from the CPU mask,
        Print_String    Rmvd            ;  then fall through to here.
Loadall_RET:
        call    Restore_state
        call    set_int6_vector
        leave
        retf
```

Listing 9.2 *continued*

```
;-----------------------------------------------------------------
;  Verify_State:                    ; Verify that LOADALL worked
;-----------------------------------------------------------------
; This is where we land for the first test of '386 LOADALL.
; The purpose of this test is to verify that all the general
; purpose registers get loaded correctly.  Specifically, we are
; testing to verify that all segment registers contain values
; that don't correspond to the memory addresses they appear to
; be pointing to.  In other words, we are checking that the
; that the segment registers have one value, while their
; associated hidden descriptor cache registers have different
; values.
;-----------------------------------------------------------------
        cmp     eax,11111111h           ; Test EAX
        jne     @F
        cmp     ebx,22222222h           ; Test EBX
        jne     @F
        cmp     ecx,33333333h           ; Test ECX
        jne     @F
        cmp     edx,44444444h           ; Test EDX
        jne     @F
        cmp     ebp,55555555h           ; Test EBP
        jne     @F
        cmp     edi,66666666h           ; Test EDI
        jne     @F
        cmp     esi,77777777h           ; Test ESI
        jne     @F
        cmp     esp,88888888h           ; Test ESP
        jne     short @F
        mov     ax,cs                   ; Test CS
        cmp     ax,1111h
        jne     short @F
        mov     ax,ds                   ; Test DS
        cmp     ax,2222h
        jne     short @F
        mov     ax,es                   ; Test ES
        cmp     ax,3333h
        jne     short @F
        mov     ax,fs                   ; Test FS
```

Listing 9.2 *continued*

```
    cmp     ax,4444h
        jne     short @F
        mov     ax,gs                   ; Test GS
        cmp     ax,5555h
        jne     short @F
        mov     ax,ss                   ; Test SS
        cmp     ax,6666h
        jne     short @F
        cmp     dword ptr ds:[0],02020202h ; Test DS Desc Cache
        jne     short @F
        cmp     dword ptr es:[0],03030303h ; Test ES Desc Cache
        jne     short @F
        cmp     dword ptr fs:[0],04040404h ; Test FS Desc Cache
        jne     short @F
        cmp     dword ptr gs:[0],05050505h ; Test GS Desc Cache
        jne     short @F
        cmp     dword ptr ss:[0],06060606h ; Test SS Desc Cache
        jne     short @F
        mov     ax,_Data                ; Reset segment regs.
        mov     ds,ax
        mov     es,ax
        lss     esp,fword ptr cs:stack_ptr  ; Reset SS:ESP
        FARJMP  <Loadall_test>,<seg _Text>  ; Continue

;---------------------------------------------------------------
; Loadall failed the REGISTERs test.
;---------------------------------------------------------------
@@:     mov     ax,_Data
        mov     ds,ax
        mov     es,ax
        lss     esp,fword ptr cs:stack_ptr
        FARJMP  <@F>,<seg _Text>
@@:     Print_String    failed
        Print_String    RFail
        jmp     loadall_ret

;---------------------------------------------------------------
; LOADALL passed
;---------------------------------------------------------------
Loadall_test:
        Print_String    passed
```

Listing 9.2 *continued*

```
;-------------------------------------------------------------
;
; TEST2:  Access extended memory while in real mode (@ 2M).
;
;  (1) Fill in LOADALL structure with "reasonable" values
;  (2) Save contents of extended memory
;  (3) Write data pattern in extended memory
;  (4) Set LOADALL registers used by this test
;  (5) LOADALL
;  (6) Verify results
;  (7) Restore original data in extended memory
;
;-------------------------------------------------------------
; In this test, I'll access extended memory while in real mode.
; I'll assume the computer has at least 2M of memory.  By
; assuming 2M, I don't need to enable A20 on the CPU bus, since
; memory @ 2M doesn't assert A20.
;-------------------------------------------------------------
; Fill LOADALL structure with more reasonable values.
;-------------------------------------------------------------
        mov     Loadall_tbl._EAX,0
        mov     Loadall_tbl._EBX,0
        mov     Loadall_tbl._ECX,0
        mov     Loadall_tbl._EDX,0
        mov     Loadall_tbl._EBP,0
        mov     Loadall_tbl._ESP,esp
        mov     Loadall_tbl._CS,cs
        mov     Loadall_tbl._DS,ds
        mov     Loadall_tbl._ES,es
        mov     Loadall_tbl._FS,8000h
        mov     Loadall_tbl._GS,8000h
        mov     Loadall_tbl._SS,ss

;-------------------------------------------------------------
; Load segment descriptor cache registers:
;  DS=ES = _DATA segment
;  FS=GS = 80000h (8000:0)
;  SS    = STACK segment
;-------------------------------------------------------------
        mov     ax,ds
        movzx   eax,ax
        shl     eax,4
        mov     bx,ss
        movzx   ebx,bx
        shl     ebx,4
        mov     Loadall_tbl.DS_Desc._addr,eax
        mov     Loadall_tbl.ES_Desc._addr,eax
        mov     Loadall_tbl.FS_Desc._addr,80000h;
        mov     Loadall_tbl.GS_Desc._addr,80000h
        mov     Loadall_tbl.SS_Desc._addr,ebx
```

Listing 9.2 *continued*

```
;--------------------------------------------------------------
; Slip into protected mode to save the contents of memory @ 2M,
; and write a test pattern into that memory.
;--------------------------------------------------------------
        Print_String    Test_2
        lgdt    fword ptr Gdt_386       ; Load the GDT
        mov     ebx,cr0                 ; enter protected mode
        or      bl,1
        mov     cr0,ebx
        FARJMP  <@F>,<CSEG3-Gdt_386>

@@:     mov     dx,DSEG3-Gdt_386        ; Get segment selector
        mov     ds,dx                   ;  for DS @ 2M
        mov     cx,400h / 4
        mov     si,0
        mov     di,offset Mem_buffer
        rep     movsd                   ; Save memory
        mov     eax,5aa5aa5ah           ; Test pattern
        mov     cx,400h / 4
        mov     di,0
        mov     es,dx                   ; ES = 2M
        rep     stosd                   ; Store test pattern
        and     bl,not 1                ; exit protected mdoe
        mov     cr0,ebx
        FARJMP  <@F>,<_TEXT>

;--------------------------------------------------------------
; Set LOADALL-image registers used by this test
;--------------------------------------------------------------
@@:     mov     ax,_data
        mov     ds,ax
        mov     es,ax
        mov     Loadall_tbl._EAX,5aa5aa5ah      ; Test pattern
        mov     Loadall_tbl._ECX,400h / 4       ; # of DWORDS
        mov     Loadall_tbl._ESI,offset Mem_buffer
        mov     Loadall_tbl._EDI,0h             ; ES:0
        mov     Loadall_tbl._EIP,offset @F      ; EIP
        mov     Loadall_tbl.ES_Desc._Addr,200000h       ; 2M
        mov     edi,offset loadall_tbl
        LOADALL
```

Listing 9.2 *continued*

```
;---------------------------------------------------------------
; Check memory for a match of the pattern
;---------------------------------------------------------------
@@:     repz    scasd                   ; data match?
        mov     bx,_Data                ; ES still points to 2M,
        mov     ds,bx                   ;  so I can reset DS and
        mov     cx,400h / 4             ;  restore the original
        mov     di,0                    ;  contents @ 2M without
        rep     movsd                   ;  changing ES.
        mov     es,bx                   ; Set ES to real mode
        lss     esp,fword ptr cs:stack_ptr      ; Restore SS:ESP
        FARJMP  <@F>,<_TEXT>                ; Must jump to R/W CS
@@:     jz      short @Test2_Pass

        PRINT_STRING    failed
        jmp     short @Test3

@Test2_Pass:
        PRINT_STRING    passed

;---------------------------------------------------------------
;
; TEST3:  Test that the Present bit gets loaded w/out exception,
;         but when a segment is accessed INT13 get generated.
;
;  (1) Load GDT & IDT pointers into LOADALL table
;  (2) Set PM bit, CS selector, EIP offset, clear P bit
;  (3) LOADALL
;  (4) Verify Results
;
;------------------------------------------------------------------------------
@Test3: Print_String    Test_3

;---------------------------------------------------------------
; Initialize the GDT & IDT descriptor cache
;---------------------------------------------------------------
        mov     eax,Loadall_tbl.DS_Desc._Addr
        mov     ebx,eax
        mov     Loadall_tbl.ES_Desc._Addr,eax
        add     eax,offset Gdt_386
        add     ebx,offset Idt_386
        mov     Loadall_tbl.GDT_Desc._Addr,eax
        mov     Loadall_tbl.GDT_Desc._Limit,Gdt3_len-1
        mov     Loadall_tbl.Idt_Desc._Addr,ebx
        mov     Loadall_tbl.Idt_Desc._Limit,Idt3_len-1

    or      Loadall_tbl._CR0,1                  ; set PM bit
        mov     Loadall_tbl._ESI,0
        mov     Loadall_tbl._CS,CSEG3-Gdt_386   ; CS selector
        mov     Loadall_tbl._EIP,offset @Test_Present
        and     Loadall_tbl.GS_Desc._Type,7fh   ; clear Present
        mov     edi,offset loadall_tbl          ;  bit in GS
        LOADALL
```

Listing 9.2 *continued*

```
@Test_Present:
        mov     si,gs:[si][2]           ; choose this because it's a 4-byte instr.
        Lidt    fword ptr RM_IDT3_Ptr   ; Restore real-mode IDT
        mov     eax,cr0                 ; Exit protected mode.
        and     al,not 1
        mov     cr0,eax
        FARJMP  <@F>,<_TEXT>            ; Return to R/W, real-mode code segment

@@:     test    si,1                    ; pass test?
        jnz     short @F                ; yep
        Print_String    failed
        jmp     short @Test4

@@:     Print_String    passed

;-----------------------------------------------------------------
;
; TEST4:  Test 32-bit Protected Mode operation
;
;  (1) Set LOADALL operand size, PM bits, and EIP offset
;  (2) LOADALL
;  (3) Verify Results
;
;-----------------------------------------------------------------
@Test4: Print_String    Test_4

;-----------------------------------------------------------------
; Perform 32-bit protected mode test
;-----------------------------------------------------------------
        or      Loadall_tbl.CS_Desc._CS32,40h   ; CS32 bit
        mov     Loadall_tbl._EDI,offset Loadall_tbl
        mov     Loadall_tbl._EIP,offset PM32
        LOADALL
```

Listing 9.2 *continued*

```
;---------------------------------------------------------------
; This test uses a simple technique to determine if we are in
; 32-bit mode or 16-bit mode:
;
; In 16-bit mode, this code will be executed as:
;       mov     si,1234h
;       nop
;       nop
;
; In 32-bit mode, this code will be executed as:
;       mov     esi,90901234h   ; The two NOP's are absorbed into the 32-bit operand.
;
;---------------------------------------------------------------
;---------------------------------------------------------------
; In 16-bit mode, this code will be executed as:
;   mov     di,1234h
;       nop
;       nop
;
; In 32-bit mode, this code will be executed as:
;       mov     edi,90901234h
;---------------------------------------------------------------
PM32:   mov     si,1234h
        nop
        nop

;---------------------------------------------------------------
; Now, we need to exit 32-bit mode gracefully. In order to do that,
; we need code that will generate predictable results in both 32-bit,
; and 16-bit mode. If we are in 32-bit mode, then the following
; compiled code will behave as documented under the '32-bit mode'
; column. If we failed to enter 32-bit mode, then the following
; compiled code will behave as documented under the '16-bit mode'
; column. In order to exit gracefully, we need to know which mode
; we are in! Hence the following code to detect it!:
;
; 16-bit mode          32-bit mode
; push    ax           push    eax
; mov     ax,si        mov     eax,esi
; shr     ax,10h       shr     eax,10h ; AX=9090 if in 32-bit
;                                      ; AX=0    if in 16-bit
;
; cmp     al,90h       cmp     al,90h  ; PASS=32-bit, FAIL=16-bit
; pushf                pushf           ; save results for later
; shl     ax,10h       shl     eax,10h ; EAX=90900000 if in 32-bit
; mov     si,ax        mov     esi,eax ; ESI=90900000 if in 32-bit
;                                      ; ESI=66660000 if in 16-bit
; popf                 popf            ; restore ZF
; pop     ax           pop     eax
;
; So, the result of this code is to exit 32-bit mode if we got
; there, and to keep going in 16-bit mode if we didn't.
;---------------------------------------------------------------
```

Listing 9.2 *continued*

```
        push    ax                      ; See the above
        mov     ax,si                   ;  explanation on how
        shr     ax,10h                  ;  this code works
        cmp     al,90h
        pushf
        shl     ax,10h
        mov     si,ax
        popf
        pop     ax
        jne     short @No_PM32bitCS
        or      si,1                    ; set SI if 32-bit mode

;--------------------------------------------------------------
; 32-bit JUMP instruction, to transfer control back to a 16-bit segment
;--------------------------------------------------------------
        LONGJMP <@F>,<CSEG3-GDT_386>    ; Construct 32-bit far JMP

@No_PM32bitCS:                          ; FAR JUMP if stuck in
        FARJMP  <@F>,<CSEG3-GDT_386>        ;  16-bit mode.

@@:     Lidt    fword ptr RM_IDT3_Ptr   ; Restore real-mode IDT
        mov     eax,cr0                 ; Exit protected mode.
        and     al,not 1
        mov     cr0,eax
        FARJMP  <@F>,<_TEXT>                ; Jump to real-mode code segment
@@:     mov     ax,_Data                ; Restore real-mode
        mov     ds,ax                   ;  segment registers
        mov     es,ax
        test    si,1                    ; 32-bit mode passed?
        jnz     short @F                ; yep
        Print_String    failed
        jmp     short @Test5

@@:     Print_String    passed

;--------------------------------------------------------------
;
; TEST5:  Test 32-bit real-mode operation
;
;  (1) Clear operand size, PM bits, and set EIP offset
;  (2) LOADALL
;  (3) Verify Results
;  (4) Get the h&*@! out of 32-bit real-mode
;
;--------------------------------------------------------------
; Test 32-bit real-mode operation.  Make sure we can put the
; processor in an illegal state:  32-bit real-mode.
;--------------------------------------------------------------
@Test5: Print_String    Test_5
        and     Loadall_tbl._CR0,not 1         ; disable PM bit
        and     Loadall_tbl.CS_Desc._Type,not 8 ; make CS R/W Data segment
        mov     Loadall_tbl._EIP,offset RM32    ; Load EIP
        LOADALL
```

Listing 9.2 *continued*

```
;--------------------------------------------------------------
; This test uses the exact same technique documented in TEST4.
; See TEST4 for an explanation of these next two code sections.
;--------------------------------------------------------------
RM32:   mov     si,1234h
        nop
        nop

;--------------------------------------------------------------
; See TEST4 for an explanation of this code section.
;--------------------------------------------------------------
        push    ax                      ; See the above
        mov     ax,si                   ;  explanation on how
        shr     ax,10h                  ;  this code works
        cmp     al,90h
        pushf
        shl     ax,10h
        mov     si,ax
        popf
        pop     ax

;--------------------------------------------------------------
; Now getting out of 32-bit real mode is a bit more complicated
; than you may think.  According to Intel, the internal
; descriptor cache registers get re-loaded with default values
; each time the segment is loaded.  But the operand size bit in
; the CS descriptor cache doesn't get cleared in a long jump.
; So, to program around this, I need to either execute LOADALL
; again, or go to protected mode, jump to a 16-bit code segment,
; then go back to real mode.  I think I will do the latter.
;--------------------------------------------------------------
        jne     short @RM32_Fail        ; Do something else
        mov     eax,cr0                 ; Enter protected mode
        or      al,1
        mov     cr0,eax
        LONGJMP <@F>,<CSEG3-GDT_386>    ; To 16-bit CS

@@:     Lidt    fword ptr RM_IDT3_Ptr   ; Reload real-mode IDT
        and     al,not 1                ; Exit protected mode
        mov     cr0,eax
        FARJMP  <@RM32_Pass>,<seg _text>; Back to real-mode CS

@RM32_Fail:
        Print_String    failed
        jmp     short @Test6

@RM32_Pass:
        mov     ax,_Data                ; Reload real-mode
        mov     ds,ax                   ;  segment registers
        mov     es,ax
        Print_String    passed
```

Listing 9.2 *continued*

```
;--------------------------------------------------------------
;
; TEST6:  Test Granularity bit. Test that the granularity bit in the descriptor
;         cache has no effect on the limit field of the descriptor cache. This
;         is a three-part test. First, by setting G=1 in a segment descriptor
;         cache whose limit=64k should cause an exception. Second, ES_LIMIT=16M-4.
;         This LIMIT can't ever be generated under program control. So we simply
;         need to access memory @ 16M-8 to verify this test. If an exception gets
;         generated, then the test failed. Third, try to access memory @ 16M-4.
;         If an exception ISN'T generated, then the test failed.
;
;  (1) Set Present, Granularity bits, and EIP offset
;  (2) LOADALL
;  (3) Verify results
;
;--------------------------------------------------------------
; Test Granularity bit: Test that G=1 has no effect in the descriptor
; cache register. In the GS descriptor cache, I set the limit=64k.
; If G=1 has any effect on this field, then the following memory
; access will pass. We want it to fail, and generate an exception-13.
;--------------------------------------------------------------
@Test6: Print_String    Test_6
        or      Loadall_tbl._CR0,1              ; set PM bit
        or      Loadall_tbl.GS_Desc._Type,80h   ; set P=1
        or      Loadall_tbl.GS_Desc._CS32,80h   ; set G=1
        and     Loadall_tbl.CS_Desc._CS32,0bfh  ; clear CS32 bit
        mov     Loadall_tbl._EIP,offset @Test_Gran
    LOADALL

@Test_Gran:
        mov     esi,10000h              ; should generate exc.
        mov     si,gs:[esi]
        shl     si,1                    ; save results
        mov     esi,0fffff8h            ; 16M-8
        mov     si,es:[esi]             ; shouldn't generate exc
        shl     si,1
        mov     esi,0fffffch            ; 16M-4
        mov     si,es:[esi]             ; should generate exc.
        add     esi,4
        Lidt    fword ptr RM_IDT3_Ptr   ; Restore real-mode IDT
        mov     eax,cr0
        and     al,not 1
        mov     cr0,eax
        FARJMP  <@F>,<_TEXT>

@@:     and     si,7
        cmp     si,5                    ; pass test?
        jnz     short @F                ; yep
        Print_String    failed
        jmp     short @Test7

@@:     Print_String    passed
```

Listing 9.2 *continued*

```
;---------------------------------------------------------------
; The following two expamples show how to use the debug
; registers to generate execution and data break points.  They
; are included in this program because they use LOADALL to set
; DR7.  Forgive the fact that they are poorly documented.
;---------------------------------------------------------------
;
; TEST7:  Test execution breakpoints through debug registers.
;
;  (1) Clear granularity bit, set ES_LIMIT=64k, set EIP offset
;  (2) Set breakpoint qualifiers
;  (3) LOADALL
;  (4) Generate execution breakpoints
;  (5) Verify results
;
;---------------------------------------------------------------
@Test7: Print_String    Test_7
        and     Loadall_tbl.GS_Desc._CS32,7fh    ; set G=0
        mov     Loadall_tbl.ES_Desc._Limit,0ffffh; limit=64k
        mov     Loadall_tbl._EIP,offset Test_DR0

;---------------------------------------------------------------
; Set a series of execution break points in the debug registers.
;---------------------------------------------------------------
        mov     bx,cs
        movzx   ebx,bx
        shl     ebx,4
        lea     eax,[ebx][dword ptr Test_Dr0]
        mov     dr0,eax
        lea     eax,[ebx][dword ptr Test_DR1]
        mov     dr1,eax
        lea     eax,[ebx][dword ptr Test_DR2]
        mov     dr2,eax
        lea     eax,[ebx][dword ptr Test_DR3]
    mov     dr3,eax
        mov     Loadall_tbl._DR7,0aah   ; enable all code breakpoints
        LOADALL

Test_DR0:
        nop
Test_DR1:
        nop
Test_DR2:
        nop
Test_DR3:
        nop

        Lidt    fword ptr RM_IDT3_Ptr   ; Restore real-mode IDT
        mov     eax,cr0                 ; exit protected mode
        and     al,not 1
        mov     cr0,eax
        FARJMP  <@F>,<seg _text>
```

Listing 9.2 *continued*

```
@@:     and     si,0fh
        cmp     si,0fh                  ; pass all breakpoints?
        je      short @F                ; yep
        Print_String    failed
        jmp     @Test8

@@:     Print_String    passed

;-------------------------------------------------------------
;
; TEST8:  Test byte-size data write breakpoints
;
;  (1) Set EIP offset
;  (2) Set breakpoint qualifiers
;  (3) LOADALL
;  (4) Generate data breakpoints
;  (5) Verify results
;
;-------------------------------------------------------------
@Test8: Print_String    Test_8
        mov     Loadall_tbl._EAX,55h
        mov     Loadall_tbl._ECX,40h
        mov     Loadall_tbl._EDI,offset Buffer
        mov     Loadall_tbl._EIP,offset @Test_DRBW

        mov     bx,ds
        movzx   ebx,bx
        shl     ebx,4
        lea     eax,[ebx][Dword ptr Buffer][3]
        mov     dr0,eax
        add     eax,10h
        mov     dr1,eax
        add     eax,10h
        mov     dr2,eax
        add     eax,10h
        mov     dr3,eax
        mov     Loadall_Tbl._DR7,111102aah
        LOADALL

@Test_DRBW:
    rep     stosb
        Lidt    fword ptr RM_IDT3_Ptr
        mov     ebx,cr0
        and     bl,not 1
        mov     cr0,ebx
        FARJMP  <@F>,<seg _text>

@@:     and     si,0fh
        cmp     si,0fh                  ; pass all breakpoints?
        je      short @T7P              ; yep
        Print_String    failed
        jmp     @Test9
```

Listing 9.2 *continued*

```
@T7P:   cmp     dx,4                    ; INT01 4 times?
        jne     @B
        Print_String    passed

@Test9:
        jmp     Loadall_ret

LOADALL_386     endp

;----------------------------------------------------------------
; Minimal exception 13 handler that points past a 4-byte opcode,
; and sets the lowest bit in DI before returning.
;----------------------------------------------------------------
        INT13   label   word
        push    bp
        mov     bp,sp
        add     word ptr [bp][4],4
        or      si,1
        pop     bp
        add     sp,2
        iret

;------------------------------------------------------------------------------
  INT01:        ; Intl trap routine
;------------------------------------------------------------------------------
; Interprets breakpoint type, and sets a flag
;------------------------------------------------------------------------------
        inc     dx
        push    bp
        mov     bp,sp
        mov     buffer2[bx],cx
        add     bx,2
        push    eax
        push    ebx
        push    ecx
        mov     eax,dr6
        mov     ecx,eax
        mov     ebx,dr7
        shr     ebx,10h                 ; get length encodings
        test    ah,20h                  ; debug register access attempt?
        jnz     @DR_Attempt             ; yep
        shr     al,1                    ; DR0?
        jc      @DR0
        shr     al,1                    ; DR1?
    jc      @DR1
        shr     al,1                    ; DR2?
        jc      @DR2
@DR3:   or      si,8
        and     cl,not 8
        test    bh,30h                  ; code, or data?
        jz      @Fault
        jmp     short @Trap
```

Listing 9.2 *continued*

```
@DR2:   or      si,4
        and     cl,not 4
        test    bh,03h                  ; code, or data?
        jz      @Fault
        jmp     short @Trap
@DR1:   or      si,2
        and     cl,not 2
        test    bl,30h                  ; code, or data?
        jz      @Fault
        jmp     short @Trap
@DR0:   or      si,1
        and     cl,not 1
        test    bl,03h                  ; code, or data?
        jz      @Fault
        jmp     short @Trap
@Fault: add     word ptr [bp][2],1
        mov     dr6,ecx
@Trap:  pop     ecx
        pop     ebx
        pop     eax
        pop     bp
        iret

@Dr_Attempt:
        push    bp
        add     word ptr [bp][2],3
        pop     bp
        iret

;----------------------------------------------------------------------------
;  Save_state    proc    near    ; Save the machine state before LOADALL
;----------------------------------------------------------------------------
        push    eax
        push    ds
        mov     si,0

        mov     di,offset Machine_State.GS_Desc
        mov     ax,5000h                    ; GS descriptor
        mov     ds,ax
        mov     ebx,05050505h
        movsd
        mov     dword ptr [si-4],ebx

        sub     ax,1000h                    ; FS descriptor
        sub     ebx,01010101h
        mov     si,0
        add     di,8
        mov     ds,ax
        movsd
        mov     dword ptr [si-4],ebx
```

Listing 9.2 *continued*

```
    sub     ax,2000h                         ; DS descriptor
        sub     ebx,02020202h
        mov     si,0
        add     di,8
        mov     ds,ax
        movsd
        mov     dword ptr [si-4],ebx
        add     ax,4000h                     ; SS descriptor
        add     ebx,04040404h
        mov     si,0
        add     di,8
        mov     ds,ax
        movsd
        mov     dword ptr [si-4],ebx
        sub     ax,3000h                     ; ES descriptor
        sub     ebx,03030303h
        mov     si,0
        add     di,14h
        mov     ds,ax
        movsd
        mov     dword ptr [si-4],ebx
        pop     ds

        mov     eax,cr0
        mov     Machine_State._CR0,eax
        pushfd
        pop     eax
        mov     Machine_State._Eflags,eax
        pop     eax
        mov     Machine_State._EDI,edi
        mov     Machine_State._ESI,esi
        mov     Machine_State._EBP,ebp
        mov     Machine_State._EBX,ebx
        mov     Machine_State._EDX,edx
        mov     Machine_State._ECX,ecx
        mov     Machine_State._EAX,eax
        mov     ax,gs
        movzx   eax,ax
        mov     Machine_State._GS,eax
        mov     ax,fs
        mov     Machine_State._FS,eax
        mov     ax,ds
        mov     Machine_State._DS,eax
        mov     ax,es
        mov     Machine_State._ES,eax
        ret
Save_state      endp
```

Listing 9.2 *continued*

```
;------------------------------------------------------------------
;   Restore_state proc    near
;   Restore the machine state after LOADALL
;------------------------------------------------------------------
        mov     ax,_data
        mov     ds,ax
        mov     ax,5000h                ; GS
        mov     es,ax
        mov     si,offset Machine_State.GS_Desc
        mov     di,0
        movsd
        sub     ax,1000h                ; FS
    add     si,8
        mov     di,0
        mov     es,ax
        movsd
        sub     ax,2000h                ; DS
        add     si,8
        mov     di,0
        mov     es,ax
        movsd
        add     ax,4000h                ; SS
        add     si,8
        mov     di,0
        mov     es,ax
        movsd
        sub     ax,3000h                ; ES
        add     si,14h
        mov     di,0
        mov     es,ax
        movsd
        mov     eax,Machine_State._ES
        mov     es,ax
        mov     eax,Machine_State._DS
        mov     ds,ax
        mov     eax,Machine_State._FS
        mov     fs,ax
        mov     eax,Machine_State._GS
        mov     gs,ax
        mov     eax,Machine_State._Eflags
        push    eax
        popfd
        mov     eax,Machine_State._CR0
        mov     cr0,eax
        mov     eax,Machine_State._EAX
        mov     ecx,Machine_State._ECX
        mov     edx,Machine_State._EDX
        mov     ebx,Machine_State._EBX
        mov     ebp,Machine_State._EBP
        mov     esi,Machine_State._ESI
        mov     edi,Machine_State._EDI
        ret
Restore_State   endp
```

Listing 9.2 *continued*

```
;---------------------------------------------------------------
; Include the CPU_TYPE procedure & LOADALL test
;---------------------------------------------------------------
            Include CPU_TYPE.ASM

_text   ends

            stack   segment para stack 'stack'
            db      400h dup (0)
            stack   ends

            end     LOADALL_386

; End of File
```

Listing 9.3 EMULOAD.ASM

```
    Page    60,132
;--------------------------------------------------------------
;
; EMULOAD.ASM   Copyright (c) 1991 Robert Collins
;
;       This utility uses '386 LOADALL to emulate '286 LOADALL.
;       All 16-bit registers are zero-extended to 32-bit
;       registers.  All 24-bit physical addresses are zero-
;       extended to 32-bit registers.  '386-specific registers
;       not used in '286 LOADALL are either set to the current
;       values (Debug registers), or zeroed (segment registers).
;
;--------------------------------------------------------------
;
;       This program assumes that you have run the '386 LOADALL test
;       prior to installing this TSR. Obviously if LOADALL has been
;       removed from the '386 mask, then this program will never work.
;       Likewise, it is easier for me to document the need to run the
;       LOADALL test program, than to incorporate it into this code.
;
;--------------------------------------------------------------
;
;       EMULOAD returns ERROR codes to DOS that can be intercepted
;       by the batch file command 'IF ERRORLEVEL'. The following
;       ERRORLEVEL codes are generated by this program:
;         0 = EMULOAD driver now installed in memory
;         1 = Attempted removal of the EMULOAD driver from memory
;             failed because EMULOAD was not in already in memory.
;         2 = The EMULOAD driver was already in memory when an
;             attempt was made to install it again.
;         3 = Bogus command line argument(s).
;         4 = Help requested.
;         5 = The EMULOAD driver was sucessfully removed from memory.
;         6 = Can't install the EMULOAD driver because this computer isn't an 80386.
;
;--------------------------------------------------------------
;
;       Compilation instructions:
;               MASM EMULOAD;           (MASM 5.1)
;               LINK EMULOAD;
;               EXE2BIN EMULOAD EMULOAD.COM
;               DEL EMULOAD.EXE
;
;       The resultant EMULOAD.COM file is 1473 bytes, while the
;       TSR portion is 1072 bytes.
;
;--------------------------------------------------------------
; Compiler directives
;--------------------------------------------------------------
        Title   EMULOAD
        .radix  16
        .8086
```

Listing 9.3 *continued*

```
;------------------------------------------------------------------
; Interrupt vector segment
;------------------------------------------------------------------
ABS0    segment at 0
        org 6*4
        INT_6   dd      ?

        org 800h
        Loadall_286     dd      ?
ABS0    ends

;------------------------------------------------------------------
; Structure definitions
;------------------------------------------------------------------
Desc_cache2     STRUC                   ; 80286 Descriptor cache
        A15_A00 dw      ?               ;  register layout.
        A23_A16 db      ?
        _Type2  db      ?
        _Limit2 dw      ?
Desc_cache2     ENDS

Desc_cache3     STRUC                   ; 80386 Descriptor cache
        _Access db      0               ;  register layout
        _Type   db      ?
        _CS32   db      0
                db      0
        _Addr   dd      ?
        _Limit  dd      ?
Desc_cache3     ENDS
```

Listing 9.3 *continued*

```
Loadall_struc2  STRUC                           ; 80286 LOADALL table
                        dw      3 dup (?)       ; RESERVED
        _286Msw         dw      ?               ; MSW
                        dw      7 dup (?)       ; RESERVED
        _286Tr          dw      ?               ; TR
        _Flags          dw      ?               ; FLAGS
        _286Ip          dw      ?               ; IP
        _286Ldt         dw      ?               ; LDT
        _286Ds          dw      ?               ; DS
        _286Ss          dw      ?               ; SS
        _286Cs          dw      ?               ; CS
        _286Es          dw      ?               ; ES
        _286Di          dw      ?               ; DI
        _286Si          dw      ?               ; SI
        _286Bp          dw      ?               ; BP
        _286Sp          dw      ?               ; SP
        _286Bx          dw      ?               ; BX
        _286Dx          dw      ?               ; DX
        _286Cx          dw      ?               ; CX
        _286Ax          dw      ?               ; AX
        ES_Desc286      dw      3 dup (?)       ; ES Desc. Cache
        CS_Desc286      dw      3 dup (?)       ; CS Desc. Cache
        SS_Desc286      dw      3 dup (?)       ; SS Desc. Cache
        DS_Desc286      dw      3 dup (?)       ; DS Desc. Cache
        Gdt_Desc286     dw      3 dup (?)       ; GDTR
        Ldt_Desc286     dw      3 dup (?)       ; LDTR
        Idt_Desc286     dw      3 dup (?)       ; IDTR
    TSS_Desc286     dw      3 dup (?)           ; TSSR
Loadall_Struc2  ENDS
```

Listing 9.3 *continued*

```
Loadall_struc3  STRUC
        _Cr0            dd      ?               ; EAX
        _Eflags         dd      ?               ; EFLAGS
        _Eip            dd      ?               ; EIP
        _Edi            dd      ?               ; EDI
        _Esi            dd      ?               ; ESI
        _Ebp            dd      ?               ; EBP
        _Esp            dd      ?               ; ESP
        _Ebx            dd      ?               ; EBX
        _Edx            dd      ?               ; EDX
        _Ecx            dd      ?               ; ECX
        _Eax            dd      ?               ; EAX
        _Dr6            dd      ?               ; DR6
        _Dr7            dd      ?               ; DR7
        _Tr             dd      ?               ; TR
        _Ldt            dd      ?               ; LDT
        _Gs             dd      ?               ; GS
        _Fs             dd      ?               ; FS
        _Ds             dd      ?               ; DS
        _Ss             dd      ?               ; SS
        _Cs             dd      ?               ; CS
        _Es             dd      ?               ; ES
        TSS_Desc        dd      3 dup (?)       ; TSSR
        IDT_Desc        dd      3 dup (?)       ; IDTR
        Gdt_Desc        dd      3 dup (?)       ; GDTR
        Ldt_Desc        dd      3 dup (?)       ; LDTR
        GS_Desc         dd      3 dup (?)       ; GS Desc. Cache
        FS_Desc         dd      3 dup (?)       ; FS Desc. Cache
        DS_Desc         dd      3 dup (?)       ; DS Desc. Cache
        SS_Desc         dd      3 dup (?)       ; SS Desc. Cache
        CS_Desc         dd      3 dup (?)       ; CS Desc. Cache
        ES_Desc         dd      3 dup (?)       ; ES Desc. Cache
                        dd      0ah dup (?)     ; RESERVED
Loadall_Struc3  ENDS

INT_VEC         STRUC
        int_offset      dw      ?
        int_segment     dw      ?
INT_VEC         ENDS

;-----------------------------------------------------------------
; Equate definitions
;-----------------------------------------------------------------
        LOADALL286      equ     050fh
        CRLF            equ     <0dh,0ah>
        CRLF$           equ     <0dh,0ah,'$'>
        INT6            equ     [bp-4]
```

Listing 9.3 *continued*

```
;-------------------------------------------------------------
; Macro definitions
;-------------------------------------------------------------
        LOADALL_386     MACRO
                db      0fh,07h
        ENDM

        PRINT_STRING    MACRO   MSG_NAME
        mov     ah,9
        mov     dx,offset MSG_NAME
        int     21h
        ENDM

_TEXT   SEGMENT PARA PUBLIC 'CODE'
        Assume  CS:_TEXT, DS:_TEXT, ES:_TEXT, SS:_TEXT
        Org     100h
        .386p
;-------------------------------------------------------------
  Emulate_286_Loadall   Proc    Far
;-------------------------------------------------------------
        jmp     EMULOAD         ; goto beginning instruction

Align 4
;-------------------------------------------------------------
; Local Data
;-------------------------------------------------------------
Loadall_tbl     Loadall_Struc3 <>

emuload_msg     db      "80286 LOADALL EMULATOR utility.",CRLF
                db      "Version 1.0 Only for 80386 computers."
                db      CRLF
                db      "Copyright (c) 1991 Robert Collins."
                db      CRLF$
emu_msg_len     equ     $-emuload_msg

align 4
;-------------------------------------------------------------
; TSR Code begins here as an INT06 replacement.
;-------------------------------------------------------------
Int06:  push    bp
        mov     bp,sp
        push    si
        push    ds
        lds     si,[bp][2]              ; get CS:IP of bogus opcode
        cmp     word ptr [si],LOADALL286; was it LOADALL?
        jne     @Not_LOADALL            ; nope
        mov     di,0
        mov     ds,di
        mov     di,cs
        mov     es,di
        mov     edi,offset Loadall_tbl
```

Listing 9.3 *continued*

```
Assume  DS:ABS0, ES:_TEXT, SS:NOTHING
;---------------------------------------------------------------
; Convert 80286 registers to 80386 counterparts.  The sequencing
; order follows the 80386 LOADALL table.
;---------------------------------------------------------------
; While mapping MSW to CR0, bit5 in CR0 is documented as
; RESERVED on the ’386 DX, and ’1’ on the ’386 SX.  Bit6 is
; defined as ’NE’ (Numeric Exception) on the ’486.  If we wanted
; this code to work on the ’486, then we should mask the lower
; nibble of MSW with CR0.  But the ’486 doesn’t have LOADALL,
; so this isn’t necesary.  Next consider the Reserved bit5 on
; the ’386 DX.      Since LOADALL completely redefines the CPU
; state, it is safe to clear this reserved bit instead of
; masking it with MSW.
;---------------------------------------------------------------
        mov     eax,cr0                 ; MSW   --> CR0
        mov     ax,Loadall_286._286Msw
        mov     Loadall_tbl._CR0,eax
        movzx   eax,Loadall_286._Flags  ; FLAGS --> EFLAGS
        mov     Loadall_tbl._EFLAGS,eax

;---------------------------------------------------------------
; Hereafter MOVZX isn’t needed because the upper 16-bits are
; guaranteed to be 0.
;---------------------------------------------------------------
        mov     ax,Loadall_286._286IP   ; IP    --> EIP
        mov     Loadall_tbl._EIP,eax
        mov     ax,Loadall_286._286DI   ; DI    --> EDI
        mov     Loadall_tbl._EDI,eax
        mov     ax,Loadall_286._286SI   ; SI    --> ESI
        mov     Loadall_tbl._ESI,eax
        mov     ax,Loadall_286._286BP   ; BP    --> EBP
        mov     Loadall_tbl._EBP,eax
        mov     ax,Loadall_286._286SP   ; SP    --> ESP
        mov     Loadall_tbl._ESP,eax
        mov     ax,Loadall_286._286BX   ; BX    --> EBX
        mov     Loadall_tbl._EBX,eax
        mov     ax,Loadall_286._286DX   ; DX    --> EDX
        mov     Loadall_tbl._EDX,eax
        mov     ax,Loadall_286._286CX   ; CX    --> ECX
        mov     Loadall_tbl._ECX,eax
        mov     ax,Loadall_286._286AX   ; AX    --> EAX
        mov     Loadall_tbl._EAX,eax
```

Listing 9.3 *continued*

```
;---------------------------------------------------------------
; DR6 & DR7 aren't in the '286, so let's use the current values.
; By keeping the current values, guarantees that any ICE
; breakpoints, or debug register breakpoints are preserved.
; (ICE breakpoints use (at least) the upper two of the
; 'RESERVED' bits in DR7.
;---------------------------------------------------------------
        mov     eax,dr6                 ; Keep DR6
        mov     Loadall_tbl._DR6,eax
        mov     eax,dr7                 ; Keep DR7
        mov     Loadall_tbl._DR7,eax

        movzx   eax,Loadall_286._286TR  ; TR    --> TR
        mov     Loadall_tbl._TR,eax
        mov     ax,Loadall_286._286LDT  ; LDT   --> LDT
        mov     Loadall_tbl._LDT,eax

;---------------------------------------------------------------
; FS & GS aren't in the '286, so let's zero them out.
;---------------------------------------------------------------
        xor     ax,ax
        mov     Loadall_tbl._GS,eax     ; Clear GS
        mov     Loadall_tbl._FS,eax     ; Clear FS

        mov     ax,Loadall_286._286DS   ; DS    --> DS
        mov     Loadall_tbl._DS,eax
        mov     ax,Loadall_286._286SS   ; SS    --> SS
    mov     Loadall_tbl._SS,eax
        mov     ax,Loadall_286._286CS   ; CS    --> CS
        mov     Loadall_tbl._CS,eax
        mov     ax,Loadall_286._286ES   ; ES    --> ES
        mov     Loadall_tbl._ES,eax

;-----------------------------------------------------------
; Convert '286 descriptor cache register entries to '386
; format.
;-----------------------------------------------------------
        mov     esi,offset Loadall_286.TSS_Desc286
        mov     edi,offset Loadall_tbl.TSS_Desc
        call    CVT_Desc
        mov     esi,offset Loadall_286.IDT_Desc286
        mov     edi,offset Loadall_tbl.IDT_Desc
        call    CVT_Desc
        mov     esi,offset Loadall_286.GDT_Desc286
        mov     edi,offset Loadall_tbl.GDT_Desc
        call    CVT_Desc
        mov     esi,offset Loadall_286.LDT_Desc286
        mov     edi,offset Loadall_tbl.LDT_Desc
        call    CVT_Desc
```

Listing 9.3 *continued*

```
;--------------------------------------------------------
; Fill in FS & GS descriptor cache entires with 0.
;--------------------------------------------------------
        mov     Loadall_tbl.GS_Desc._Type,93h
        mov     Loadall_tbl.GS_Desc._Addr,0
        mov     Loadall_tbl.GS_Desc._Limit,0ffffh
        mov     Loadall_tbl.FS_Desc._Type,93h
        mov     Loadall_tbl.FS_Desc._Addr,0
        mov     Loadall_tbl.FS_Desc._Limit,0ffffh

;--------------------------------------------------------
; Convert '286 descriptor cache register entries to '386 format.
;--------------------------------------------------------
        mov     esi,offset Loadall_286.DS_Desc286
        mov     edi,offset Loadall_tbl.DS_Desc
        call    CVT_Desc
        mov     esi,offset Loadall_286.SS_Desc286
        mov     edi,offset Loadall_tbl.SS_Desc
        call    CVT_Desc
        mov     esi,offset Loadall_286.CS_Desc286
        mov     edi,offset Loadall_tbl.CS_Desc
        call    CVT_Desc
        mov     esi,offset Loadall_286.ES_Desc286
        mov     edi,offset Loadall_tbl.ES_Desc
        call    CVT_Desc
        mov     edi,offset Loadall_tbl
        LOADALL_386
        HLT                             ; This instruction never gets executed

@Not_LOADALL:
        pop     ds
        pop     si
        pop     bp

Orig_int06:
    jmp     far ptr INT_6
Emulate_286_Loadall     endp
```

Listing 9.3 *continued*

```
;-------------------------------------------------------------
;  CVT_Desc       proc    near    ; Convert '286 descriptor table
;                                 ; cache register format to '386 format.
;-------------------------------------------------------------
; Input:   DS:ESI = Pointer to '286 descriptor cache entry
;          DS:EDI = Pointer to '386 descriptor cache entry
; Output:  None
; Register(s) modified:  EAX, EBX, ECX
;-------------------------------------------------------------
        mov     eax,[esi]               ; get 24-bit base & access rights
        mov     ebx,eax                 ; make a copy
        movzx   ecx,[esi]._Limit2       ; get 16-bit limit
        rol     eax,8                   ; put access in AL
        and     ebx,00ffffffh           ; make 24-bit address
        mov     ES:[edi]._Type,al       ; store Access
        mov     ES:[edi]._Addr,ebx      ; store Address
        mov     ES:[edi]._Limit,ecx     ; store Limit
        ret
CVT_Desc        endp

TSR_End label   word
;-------------------------------------------------------------
; End of TSR program
;-------------------------------------------------------------

;-------------------------------------------------------------
; Local DATA used for initialization code only.
;-------------------------------------------------------------
bogus_msg1      db      "Unrecognized command line argument."
                db       CRLF$
bogus_msg2      db      "Not 80386 computer.",7,CRLF$

driver_msg1     db      "Resident driver installed."
                db       CRLF$
driver_msg2     db      "Resident driver already installed."
                db       7,CRLF$
driver_msg3     db      "Resident driver removed from memory."
                db       CRLF$
driver_msg4     db      "Resident driver was not already "
                db      "installed",7,CRLF$
help_msg        db       CRLF
                db      "Syntax:  EMULOAD",CRLF
                db      "         EMULOAD -R (to remove from "
                db      "memory)",CRLF$

ASSUME  DS:_TEXT
;-------------------------------------------------------------
;  EMULOAD        proc    near    ; Beginning of initialization code
;                                 ; as the NON-TSR part of the program.
;-------------------------------------------------------------
        cld                             ; clear direction flag
    Print_String    emuload_msg     ; Print initialization message.
```

Listing 9.3 *continued*

```
;---------------------------------------------------------------
; Check CPU type
;---------------------------------------------------------------
        call    CPU_TYPE        ; Get CPU type
        and     al,0fh          ; mask out CPU sub-type
        cmp     al,3            ; 80386?
        jz      short @F        ; yes
        Print_String    Bogus_msg2      ; Not 80386 computer
        mov     ax,4c06h                ; set function to DOS
        int     21h                     ; exit to DOS

;---------------------------------------------------------------
; Check command line argument
;---------------------------------------------------------------
@@:     xor     ax,ax                   ; clear AX
        mov     si,80h                  ; get start of PSP
        lodsb                           ; get command line len.
        or      ax,ax                   ; Any command line args?
        jz      short Installed?        ; nope
        mov     cx,ax                   ; put into counter
        mov     di,si                   ;
        mov     al,' '                  ; skip past superfluous
        repz    scasb                   ;   blank characters
        cmp     byte ptr [di],0dh       ; are we at the end?
        jz      short Installed?        ; yep
        cmp     byte ptr [di-1],'-'     ; check if it's a switch
        jnz     short @F                ; if not, then error
        mov     si,di                   ; get pointer
        lodsb                           ; get cmd line switch
        cmp     al,'r'                  ; remove driver?
        jz      short remove_driver     ; yep
        cmp     al,'R'                  ; remove driver?
        jz      short remove_driver     ; go remove driver
        cmp     al,'?'                  ; help?
        jnz     short @F                ; nope
        Print_String    help_msg        ; Print help message
        mov     ax,4c04h                ; set return code
        int     21h                     ; exit to DOS

;---------------------------------------------------------------
; Bogus command line argument
;---------------------------------------------------------------
@@:     Print_String    bogus_msg1      ; Invalid command line
        mov     ax,4c03h                ; set function code
        int     21h                     ; exit to DOS
```

Listing 9.3 *continued*

```
;----------------------------------------------------------------
; Remove driver from memory
;----------------------------------------------------------------
remove_driver:
        call    check_installed         ; Driver installed?
        jnz     short @F                ; driver not installed
        mov     bp,sp                   ; create stack frame
        push    ds                      ; save (DS)
        mov     dx,ABS0                 ; get bottom of memory
        mov     ds,dx                   ; make segment register

ASSUME      DS:ABS0, ES:_TEXT
;----------------------------------------------------------------
; Restore original INT6 vector
;----------------------------------------------------------------
; We can restore the original INT6 by getting the vector from
; our current memory resident driver -- not the DS from the
; code now executing.  The original DS is the same as the code
; segment for our EMULOAD driver.  Hence we only need to get
; the original segment value from the memory resident image.
; And we get this by looking at the segment for INT6!
;----------------------------------------------------------------
        mov     es,int_6.int_segment            ; Original DS
        mov     ax,es:orig_int06[1].int_offset  ; Original INT6
        mov     bx,es:orig_int06[1].int_segment ;    "      "
        mov     int_6.int_offset,ax             ; Restore orig.
        mov     int_6.int_segment,bx            ;  INT6

;----------------------------------------------------------------
; Free memory pointed to by ES
;----------------------------------------------------------------
        mov     ah,49h                  ; DOS FREE_MEM function
        int     21h                     ; free allocated memory
        mov     ds,[bp-2]               ; get original (DS)

ASSUME  DS:_TEXT
;----------------------------------------------------------------
; Now split with TSR removed from memory.
;----------------------------------------------------------------
        Print_String    driver_msg3     ; Driver removed
        mov     ax,4c05h                ; set function to DOS
        int     21h                     ; exit to DOS

;----------------------------------------------------------------
; If EMULOAD was not in memory, then come here and split with
; the error code.
;----------------------------------------------------------------
@@:     Print_String    driver_msg4     ; Driver not installed
        mov     ax,4c01h                ; set function to DOS
        int     21h                     ; exit to DOS
```

Listing 9.3 *continued*

```
;--------------------------------------------------------------
; Check for driver already installed
;--------------------------------------------------------------
Installed?:
        call    check_installed         ; check if driver is
        jnz     short @F                ;  already installed?

;--------------------------------------------------------------
; Driver already installed
;--------------------------------------------------------------
        Print_String    driver_msg2     ; Driver already inst.
        mov     ax,4c02h                ; set function to DOS
        int     21h                     ; exit to DOS

;--------------------------------------------------------------
; Driver not yet installed
;--------------------------------------------------------------
@@:     Print_String    driver_msg1     ; Driver now installed

;--------------------------------------------------------------
; Install driver into memory
;--------------------------------------------------------------
        xor     dx,dx                   ; Point to INT. vectors
        mov     ds,dx                   ; complete the move
ASSUME  ds:ABS0

;--------------------------------------------------------------
; Chain to INT6 by replacing and saving the original INT6
; vector.
;--------------------------------------------------------------
        mov     ax,int_6.int_offset             ; Orig. offset
        mov     bx,int_6.int_segment            ; Orig. segment
        mov     orig_int06[1].int_offset,ax     ; save old INT6
        mov     orig_int06[1].int_segment,bx    ;  vector.

;----------------------------------------------------------------
; Now replace the original INT6 vector.
;--------------------------------------------------------------
        mov     dx,offset cs:int06      ; Get new INT6 vector
        mov     int_6.int_offset,dx     ;  as CS:INT6
        mov     int_6.int_segment,cs    ;

ASSUME  DS:_TEXT
```

Listing 9.3 *continued*

```
;---------------------------------------------------------------
; Terminate and Stay Resident
;---------------------------------------------------------------
        mov     dx,cs                    ; make DS=CS
        mov     ds,dx
        mov     es,ds:[2ch]              ; get DOS env. segment
        mov     ah,49h                   ; release memory func.
        int     21h                      ; release memory
        mov     dx,offset tsr_end        ; get ending address
        shr     dx,4                     ; divide by 16
        adc     dx,1                     ; check for remainder; add 1
        mov     ax,3100h                 ; set return code to DOS
        int     21h
EMULOAD         endp

ASSUME  ES:ABS0
;---------------------------------------------------------------
; Check to see if the EMULOAD driver is installed in memory.
; It is possible to check if a TSR program is already installed
; in memory by looking for a semaphore in the memory image.
; Luckily we can locate the memory image of our TSR by looking
; at the current INT6 vector.  The INT6 code segment is the
; segment of the TSR!  So this routine looks in this segment
; for the inital banner message:
;
;   80286 LOADALL EMULATOR utility.
;   Version 1.0 Only for 80386 computers.
;   Copyright (c) 1991 Robert Collins.
;
; If this message is found, then the TSR is in memory.  If
; another TSR has chained to the same INT6 vector, this
; technique will fail to find EMULOAD, as it very well should!
;---------------------------------------------------------------
;  Check_installed       proc    near
;---------------------------------------------------------------
; Input:  None
; Output:  NZ if NOT installed
;          ZF if ALREADY installed
; Register(s) modified:  CX, SI, DI
;---------------------------------------------------------------
        push    es                       ; save (ES)
        mov     cx,ABS0                  ; get bios data segment
        mov     es,cx                    ; put in (ES)
        mov     cx,emu_msg_len           ; # of bytes to compare
        mov     si,offset emuload_msg    ; get address of message
        les     di,ES:INT_6              ; get INT6 vector
        sub     di,int06-emuload_msg     ; point to theoretical
                                         ;  start of message
        repz    cmpsb                    ; check data
        pop     es                       ; restore (ES)
        ret                              ; split
Check_installed endp
```

Listing 9.3 *continued*

```
;-----------------------------------------------------------------
; Include the CPU_TYPE procedure & LOADALL test
;-----------------------------------------------------------------
        Include CPU_TYPE.ASM

_TEXT   ends
        end     Emulate_286_LOADALL

; End of File
```

Listing 9.4 CPU_TYPE.ASM

```
        .386p
;---------------------------------------------------------------;
; CPU_Type determines the CPU type in the system.               ;
; Written by: Robert Collins                                    ;
; Input:    None                                                ;
; Output:   AX = CPU type                                       ;
;              0 = 8086/8088                                    ;
;              1 = 80186/80188                                  ;
;              2 = 80286                                        ;
;              3 = 80386                                        ;
;              4 = 80486                                        ;
;           FFFF = Unknown CPU type                             ;
; Register(s) modified:  AX, BX, CX, EDX
;---------------------------------------------------------------;
; Macro definitions                                             ;
;---------------------------------------------------------------;
; 80486 instruction macro -- because MASM 5.1 doesn't support   ;
; the 80486!                                                    ;
;---------------------------------------------------------------;
; XADD    macro                                                 ;
;       db      0fh,0C0h,0D2h   ; 80486 instruction macro
; ENDM                                                          ;
;---------------------------------------------------------------;
;  CPU_Type      proc    near                                   ;
;---------------------------------------------------------------;
; Determine the CPU type by testing for differences in the CPU  ;
; in the system.                                                ;
;---------------------------------------------------------------;
; To determine if we are a 8086/8088, or 80186/80188, test the
; value of SP after it is placed on the stack.  The algorithm
; for "PUSH SP" differs from 8086/80186 to 80286+.  The
; algorithm difference is as follows:
;
; 8086/80186                    80286+
; {                             {
;   SP    = SP - 2                TEMP  = SP
;   SS:SP = SP                    SP    = SP - 2
; }                               SS:SP = TEMP
;                               }
;
; Thus for the 8086/80186, the value of SP that gets pushed on
; the stack is the value after SP is decremented.  Hence, the
; value on the stack does not reflect the value of SP before the
; "PUSH" instruction.  Therefore, all we have to do to
; categorize the CPU as 8086/8088 or 80186/80188 is to "PUSH SP"
; and compare the value on the stack image to the value in SP.
;---------------------------------------------------------------
        xor     ax,ax           ; clear CPU type return register
        push    sp              ; save SP on stack to look at
        pop     bx              ; get SP saved on stack
        cmp     bx,sp           ; if 8086/8088 these values will differ
        jz      short @Not_8086 ; nope, must be other CPU type
```

Listing 9.4 *continued*

```
;---------------------------------------------------------------
; If this test passes, then we need some other means to differ-
; entiate between 8088/8088 and 80186/80188.  This method I will
; use comes from "80186/188, 80C186/C188 Hardware Reference
; Manual" from Intel, PN# 270788, page A-2:  "When a word write
; is performed at offset FFFFh in a segment, the 8086 will write
; one byte at offset FFFFh, and the other at offset 0, while an
; 80186 family processor will write one byte at offset FFFFh,
; and the other at offset 10000h (one byte beyond the end of the
;  segment).
;---------------------------------------------------------------
; Before we can blast a value out to FFFFh, we must save
; anything there, so we don't crash anybody else's data.
;---------------------------------------------------------------
        push    es
        mov     bx,ds           ; get original DS
        inc     bx
        mov     es,bx
        mov     bl,ds:[0]               ; get byte @ 0
        mov     bh,es:[0fff0h]          ; get byte @ 10000h
        mov     ds:[0ffffh],0aaaah      ; write signature at test location
        cmp     byte ptr ds:[0],0aah    ; 8086?
        mov     ds:[0],bl               ; restore original value
        mov     es:[0fff0h],bh
        pop     es
        je      short CPU_8086_Exit
        inc     ax
        jmp     short CPU_8086_Exit

;---------------------------------------------------------------
; When we get here, we know that we aren't a 8086/80186.  And
; since all subsequent processors will trap invalid opcodes via
; INT6, we will determine which CPU we are by trapping an
; invalid opcode.
;   We are an 80486 if:  XADD   DX,DX   executes correctly
;             80386 if:  MOV    EDX,CR0 executes correctly
;             80286 if:  SMSW   DX      executes correctly
;---------------------------------------------------------------
; Setup INT6 handler
;---------------------------------------------------------------
@Not_8086:
        enter   4,0             ; create stack frame
        mov     word ptr INT6,offset INT6_handler
        mov     INT6+2,cs
        call    set_INT6_vector ; set pointer for INT6 handler
        mov     ax,4            ; initialize CPU flag=4 (80486)
        xor     cx,cx           ; initialize semaphore
```

Listing 9.4 *continued*

```
;-----------------------------------------------------------------
; Now, try and determine which CPU we are by executing invalid
; opcodes.  The instructions I chose to invoke invalid opcodes,
; are themselves rather benign.  In each case, the chosen
; instruction modifies the DX register, and nothing else.  No
; system parameters are changed, e.g. protected mode, or other
; CPU dependant features.
;-----------------------------------------------------------------
; The 80486 instruction 'XADD' xchanges the registers, then adds
; them.  The exact syntax for a '486 compiler would be:
; XADD DX,DX.
;-----------------------------------------------------------------
        XADD    ;DX,DX          ; 80486
        jcxz    CPU_exit
        dec     ax              ; set 80386 semaphore
        inc     cx              ; CX=0

;-----------------------------------------------------------------
; For a description on the effects of the following instructions,
; look in the Intel Programmers Reference Manual's for the 80186,
; 80286, or 80386.
;-----------------------------------------------------------------
        mov     edx,cr0         ; 80386
        jcxz    CPU_exit
        dec     ax              ; set 80286 semaphore
        inc     cx              ; CX=0

        smsw    dx              ; 80286
        jcxz    CPU_exit
        sub     ax,3            ; set UNKNOWN_CPU semaphore

CPU_exit:
        call    set_INT6_vector
        leave

CPU_8086_exit:
        ret
```

Listing 9.4 *continued*

```
;----------------------------------------------------------------
; Set the INT6 vector by exchanging it with the one currently on
; the stack.
;----------------------------------------------------------------
set_INT6_vector:
        push    ds
        push    ABS0            ; save interrupt vector segment
        pop     ds              ; make DS=INT vector segment

ASSUME  DS:ABS0
        mov     dx,word ptr ds:INT_6    ; get offset of INT6
        xchg    INT6,dx                 ; set new INT6 offset
        mov     word ptr ds:INT_6,dx
        mov     dx,word ptr ds:INT_6[2] ; get segment of INT6
        xchg    INT6+2,dx               ; set new INT6 segment
        mov     word ptr ds:INT_6[2],dx
        pop     ds                      ; restore register
        ret                             ; split
ASSUME  DS:_TEXT

;----------------------------------------------------------------
; INT6 handler sets a semaphore (CX=FFFF) and adjusts the return
; address to point past the invalid opcode.
; [BP]
;----------------------------------------------------------------
INT6_handler:
        enter   0,0             ; create new stack frame
        dec     cx              ; make CX=FFFF
        add     word ptr ss:[bp][2],3   ; point past invalid
                                        ; opcode
        leave
        iret
CPU_Type        endp

; End of File
```

Chapter 10

Critical Error Handling

David L. Burki

Today's PC users have come to expect a seamless user interface. Novice and seasoned users alike expect colorful pop-up windows, pull-down menus, and context-sensitive help. They also expect the program to withstand any abuse. What happens to the beautiful input screen when the user directs the program to write to an empty disk drive? Reliability of the application is more important than aesthetics. Well crafted, professional applications should be able to withstand empty disk drives, write-protected disks, and printers out of paper, without losing data. They should also present meaningful error descriptions to assist users in understanding the cause of the error. Applications should accomplish all of this without writing all over the input screen. Unfortunately, many applications don't receive any attention to these kinds of problems. This chapter presents a means of trapping DOS critical errors and shows how to control the error message display, as well as controlling what happens to your program after the user makes a response.

Critical Errors

A DOS "critical error" happens when an interrupt *21H* function attempts to interact with a system device and fails. All DOS functions are suspended until the error condition is corrected. When such an error condition occurs, DOS invokes the critical error handler specified in the interrupt vector table (*INT 24h*). The default DOS critical error handler displays a somewhat cryptic explanation of the error and solicits a user response with the infamous "*Abort, Retry, Ignore*" prompt. The explanation usually doesn't give the user the underlying cause of the error. If the user chooses the Abort option, your program may not be in a good position to terminate without losing data.

When DOS encounters a critical error, the failing DOS function places information pertaining to the error in specific registers and restores the stack to the state it was in when the *INT 21h* function dispatcher received

OFFSET FROM SP (HEX)	SAVED REGISTER	DESCRIPTION
+00	IP	Pushed by the DOS function which executed the interrupt 24h.
+02	CS	
+04	FLAGS	
+06	AX	Registers of the application that invoked the DOS function (INT 21h) that failed. AH has the DOS function number.
+08	BX	
+0A	CX	
+0C	DX	
+0E	SI	
+10	DI	
+12	BP	
+14	DS	
+16	ES	
+18	IP	Pushed by the application program executing the interrupt 21h function that failed.
+1A	CS	
+1C	FLAGS	

Figure 10.1 *State of the stack upon entry to the critical error handler.*

control. Figure 10.1 depicts the state of the stack when the critical error handler receives control, and Figures 10.2a and 10.2b show the contents of the registers. The *IP*, *CS*, and *FLAGS* at the top of the stack are the registers pushed by the failing DOS function when it invoked the critical error interrupt. This stack configuration allows the error handler to return to the DOS function which invoked it using an *IRET* instruction. The error handler is expected to return a value in the *AL* register which indicates the

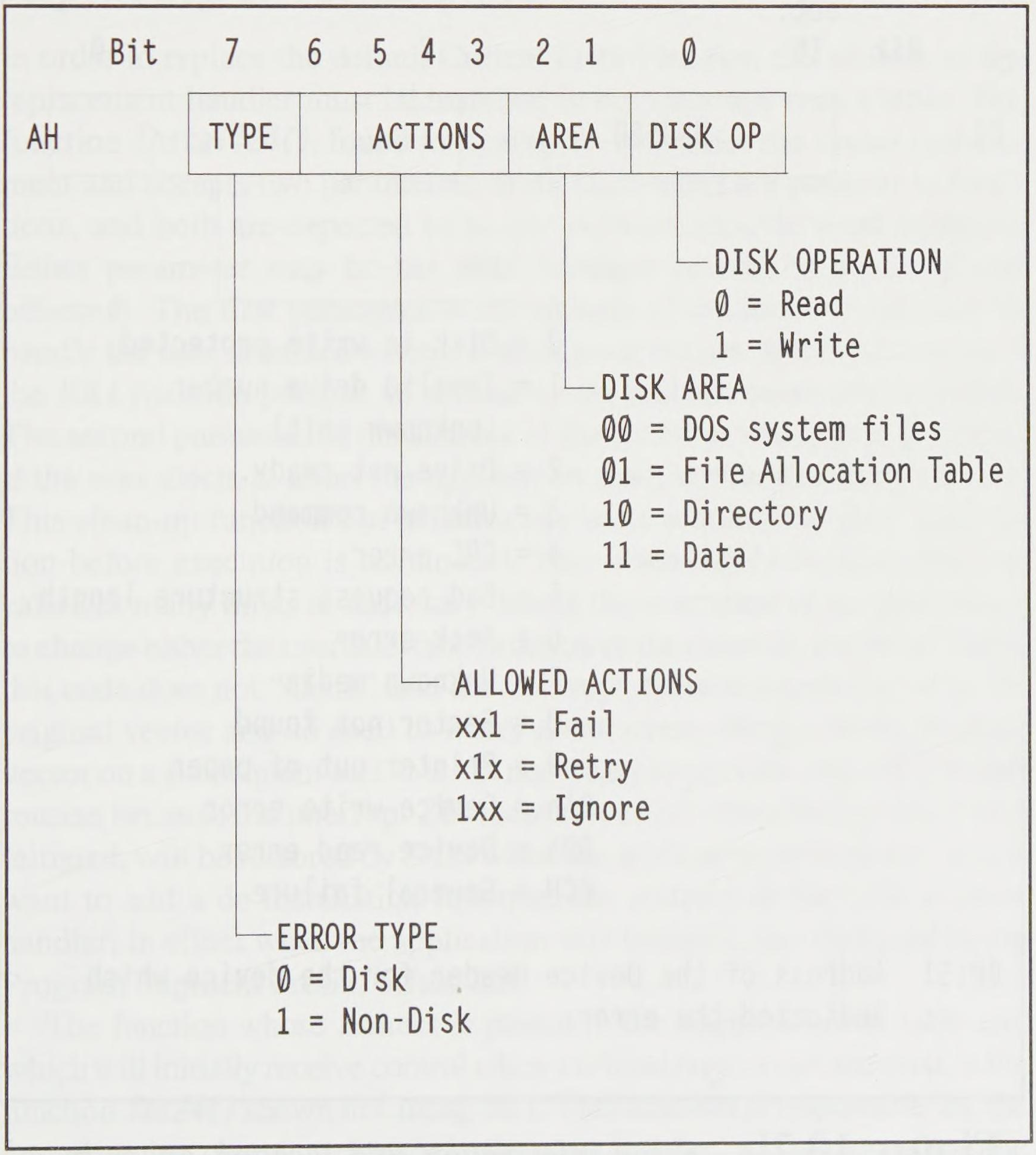
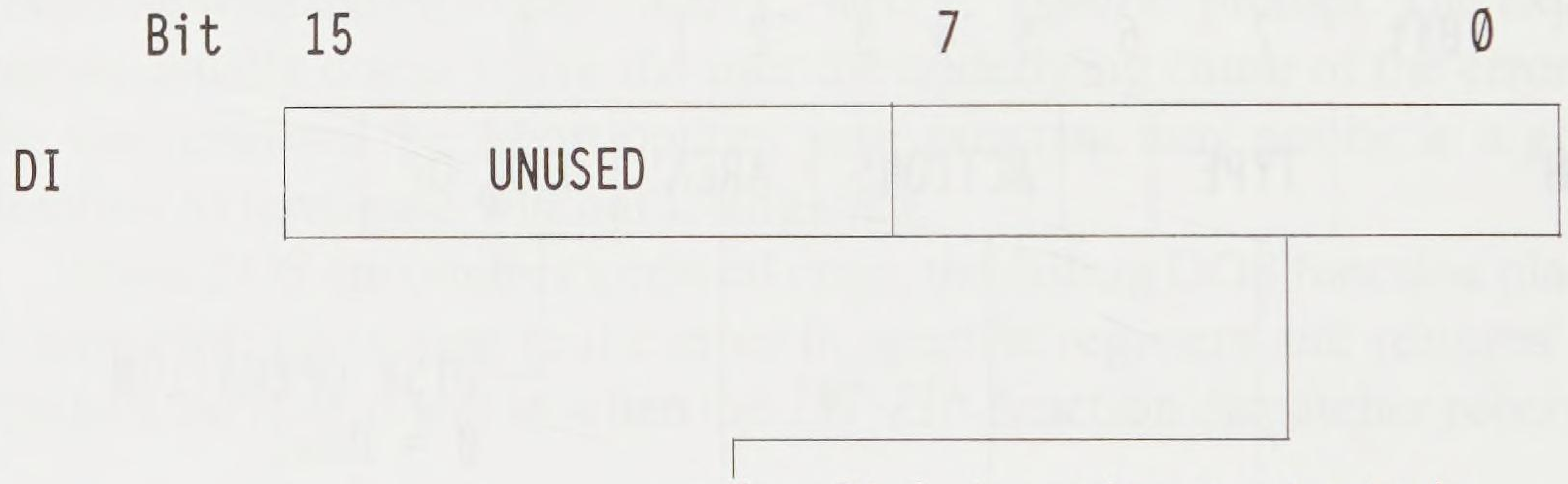

Bit 7 6 5 4 3 2 1 0
AH
TYPE
ACTIONS
AREA
DISK OP
DISK OPERATION
0 = Read
1 = Write
DISK AREA
00 = DOS system files
01 = File Allocation Table
10 = Directory
11 = Data
ALLOWED ACTIONS
xx1 = Fail
x1x = Retry
1xx = Ignore
ERROR TYPE
0 = Disk
1 = Non-Disk

Figure 10.2a *Error information and location on entry to error handler* (AH *and* AL).

users choice of the available options. When the interrupted DOS function regains control, the value in the *AL* register is examined to determine what action the interrupted function should take.

AL DRIVE NUMBER IF ERROR TYPE IS 1:
0 = A:
1 = B:
2 = C:
ect.



0 = Disk is write protected
1 = Invalid drive number (unknown unit)
2 = Drive not ready.
3 = Unknown command
4 = CRC error
5 = Bad request structure length
6 = Seek error
7 = Unknown media
8 = Sector not found
9 = Printer out of paper
0Ah = Device write error
0Bh = Device read error
0CH = General failure

BP:SI Address of the Device Header for the device which indicated the error.

Figure 10.2b *Error information and location on entry to error handler* (DI *and* BP:SI).

The following list summarizes the valid return values and their associated meaning:

`0` Ignore the error,

`1` Retry the operation,

`2` Abort,

`3` Fail (introduced in DOS v3.3).

Replacing Error Handlers

In order to replace the default Critical Error Handler, the address of the replacement handler must be installed in the interrupt vector table. The function `install24()`, found in Listing 10.1, handles the vector replacement and accepts two parameters. Both parameters are pointers to functions, and both are expected to be `far` pointers (double word address). Either parameter may be the `NULL` function pointer (segment=`0` and offset=`0`). The first parameter is the address of the function which will handle the user interface when a critical error occurs. If this parameter is the `NULL` function pointer, all critical errors will automatically be failed. The second parameter is the address of the function which is to be called if the user elects to abort the application from within the error handler. This clean-up function can perform any tasks required by your application before execution is terminated. The `install24()` function may be called as many times as necessary during the execution of an application to change either the user interface function or the clean-up function. Since this code does not "chain" into the interrupt, there is no need to save the original vector and no need to worry about overwriting a saved original vector on a subsequent call. It is not necessary to provide a de-installation routine because the interrupt `24h` vector, in effect when the program was initiated, will be restored by DOS when the application terminates. If you want to add a de-installation function, the address of the critical error handler, in effect when the application was initiated, can be found in the Program Segment Prefix, offset `12h`.

The function whose address is placed in the interrupt vector table and which will initially receive control when a critical error is encountered, is the function `int24()` shown in Listing 10.1. This function is responsible for the housekeeping chores necessary in the error handler, as well as calling the user interface and clean-up functions if required. The `int24()` function has no interaction with the user. The actual user interface routine is the function whose address was passed as the first parameter to `install24()`. By using pointers to the user interface and clean-up functions, which are called by the replacement error handler, the user interface and clean-up code can be modified without having to change any of the low level code. Changes to either routine do not affect the basic error handler housekeeping code.

Housekeeping

When the replacement error handler receives control, the user stack is in effect. In most cases, it is safe to assume there will be enough stack space available for the error handler to operate with the user stack. If there is any doubt about stack space, the error handler can be easily modified to provide its own stack. The housekeeping done by `int24()` begins by saving the current stack pointer (`SP`) in a code segment addressable variable, `save_sp`. By storing the current stack pointer on entry, you avoid having to calculate how many pushes and pops the `int24()` code has done when you prepare to exit the routine. If a user interface function is installed, indicated when the high word of `user_io` is not `0`, all the information about the error passed to the error handler is stored in a data segment accessible to the user interface code, and control is transferred to the user interface function. If no user interface function is installed, `AX` is loaded with `3`, and the jump to the label, `stabilize`, is taken. The same logic is then followed as if the user had selected the fail option.

When the user interface function has finished its dialog with the user, it is expected to return `0`, `1`, `2`, or `3` in the `AX` register, based on the user's choice of ignore, retry, abort, or fail, respectively. The `int24()` function branches to the `back_to_dos` label if the user specified ignore or retry. This branch in the logic has the least amount of work. The registers which were saved on entry to `int24()` are restored, the original stack pointer is restored, and an `IRET` returns to the DOS function which invoked the critical error handler. The value in `AL`, returned from the user interface function, tells DOS whether to ignore or retry the failed function. If the user interface function returns `2` or `3` in the `AX` register, the function continues at the label `stabilize`. The DOS *Technical Reference* states that a replacement error handler may handle the error itself and return to the instruction immediately following the `INT 21h` that experienced the error rather than returning to DOS. The *Technical Reference* continues with a warning, however: "Note that if this is done, DOS will be in an unstable state until a function call higher than `0ch` is issued; therefore, it is not recommended."

Following this advice (most of it anyway), you need to choose a DOS function which cannot encounter a critical error itself. Function `30h`, `Get DOS Version`, is an ideal choice. It cannot generate a critical error, is not likely to have been replaced by a TSR which doesn't pass the call on to DOS, and requires more than superficial processing by DOS. A logical extension to the warning provided in the *Technical Reference* indicates that since the abort option may need to use DOS functions in preparation for exit, the stabilizing call needs to be made in the abort case also. Once DOS is stabilized, you can continue with either the clean-up routine or set up to return a fail indication to the caller of the DOS function which generated the error. This method of failing is valid, even in DOS versions not directly supporting the fail option.

In order to support the fail option in versions of DOS not directly supporting the fail option, it is necessary to set the appropriate error return code in either `AL` or `AH`, depending on where the failing DOS function would normally be expected to place the return code for the caller to examine. For DOS functions greater than `36h`, the DOS error number is expected to occupy the full 16 bits of the `AX` register. Functions `0fh` through `35h` place `0ffh` in `AL`, while functions less than `0fh` place `0ffh` in `AH`. Function `36h` uses the entire `AX` register to return `0ffh`. Table 10.1 summarizes the failing return codes and their register placement by DOS function number. The section of `int24()` code beginning at the label `do_fail` restores `SP` to the value it contained when the error handler was entered, retrieves the failing DOS function number from the stack, and stores the appropriate return value in the correct eight or 16 bits of the `AX` register. The `CS`, `IP`, `FLAGS`, and `AX` registers, placed on the stack when the failing DOS function invoked the error handler, are discarded by adding eight to `SP`. The rest of the registers are then restored to the values the

Failing Function #	**Error Code Register**	**Error Code Value**
`> 36h`	`AX`	`19` through `34`
`36h`	`AX`	`-1`
`0fh` through `35h`	`AL`	`-1`
`< 0fh`	`AH`	`-1`

Table 10.1 ***Returned values by function number for fail option.***

failing DOS function saved before invoking the error handler. The `AX` register on the stack is discarded because it already contains the necessary return value. Since functions greater than `36h` expect the carry flag to be set as an indication that the value in `AX` is an error number, the error handler will always set the carry flag before failing the function. DOS functions `00h` through `36h` don't care about the state of the carry flag. The `RET 2` instruction at the end of the `do_fail` block, returns to the instruction immediately following the failed `INT 21h` instruction. The `RET 2` is used instead of an `IRET` so the `FLAGS` register, pushed by the `INT 21h` instruction, will be discarded.

When the user elects to abort the program during a critical error (the user interface function returns `2`), the jump to the label `do_abort_routine` is taken. The abort block of code restores `SP` to the value it had on entry to the error handler and then discards all the information placed on the stack by the failing DOS function, as well as the `CS`, `IP`, and `FLAGS` registers, placed on the stack by the failed `INT 21h`. Reclaiming this stack space is not mandatory, but it will give the clean-up function a bit more operating room. If a clean-up function has been installed, control is passed to it. Once the clean-up routine has completed its tasks, control is returned to `int24()`; it will then terminate the process using the DOS function `4ch`. The clean-up routine is free to make any DOS function call it needs. However, the clean-up routine shouldn't do more than is absolutely necessary to gracefully exit the program. It would lead to nasty things if the clean-up routine itself encountered a critical error, and the user elected to abort, and the clean-up routine itself encountered a critical error and...

User Interface

Now that we've covered the details of the handler, it's time to look at the user interface code. The user interface function has one severe restriction, depending on your point of view. Since DOS is not reentrant, the user interface code cannot call any DOS function higher than `0ch`, with the exception of functions `30h` (`Get DOS Version`) and `59h` (extended error reporting). Calls to other DOS functions may work as expected, but remember that the user might elect to ignore or retry the failed function. In those cases, the replacement error handler returns to DOS, and who knows what state DOS will be in if the user interface code has made DOS function calls other than those allowed. These restrictions on DOS calls prohibit the user interface from doing things like dynamic memory allocation and disk I/O. Most high level languages use DOS functions greater than `0ch` to do console I/O. For example, Microsoft C uses DOS function `40h` (write to file or device) to display text via the `printf()` function. Using `printf()` inside the user interface causes an immediate hang which requires that the system be re-booted.

The user interface function has several tasks to perform. The area of the screen where the error message will be displayed must be saved and restored. The information about the error which `int24()` saved must be deciphered and the appropriate messages constructed and displayed. User input must be accepted and validated before returning to the error handler for evaluation. The user interface routine may be written in any language, as long as you remember that DOS functions higher than `0ch` must be avoided. The mainline code for my user interface is written in C. Using a high level language makes initial development faster, the function logic is easier to follow, and enhancements are easier to code. The user interface code presented here does not check every potential error. Possible enhancements would be to obtain and display extended error information obtained using DOS function `59h` and providing a check for removable disk media. The user interface routine presented here makes the assumption that the display is in the 25 x 80 text mode. All the screen I/O routines rely on this to function properly. Since this chapter is not about doing console I/O without DOS help, there isn't a detailed description of the support functions found in Listing 10.4. The functional descriptions at the beginning of each support routine and the comments within the code should give you enough information to understand the function.

The function `user_interface()` in Listing 10.2 is the heart of the user interface. Because an application may change the active display any time during execution, the user interface code determines the active display adapter by calling the function `monitor()`. The `monitor()` function establishes the segment address of video RAM used by all the other screen I/O routines. The array, `valid_options`, used to limit the possible user responses, is initialized to `0`. Later on, elements of the array will be filled with a keyboard scan code if the associated user response is valid. The rectangular region of the display where the error message will appear is saved with a call to the function `save_window()`, and the window is

```
      Field                             Length
      ------------------------------------------

      Pointer to next header            Double Word

      Attribute*                        Word

      Pointer to device strategy
      routine                           Word

      Pointer to device interrupt
      routine                           Word

      Name/unit field                   Byte[8]


* Attribute bit fields:

     Bit                 Meaning
    ------------------------------------------------------

      15                 = 1 character device
                           0 block device
      14                 = 0 supports IOCTL
                           1 doesn't support IOCTL
      13                 For block device drivers:
                           = 1 non-IBM Format
                           = 0 IBM format
                         For character device drivers:
                           = 1 supports output until busy
                           = 0 doesn't support output until
busy
      12                 Unknown
      11                 = 1 supports removable media
                           0 doesn't support removable media
    10-7                 Reserved for DOS
```

Figure 10.3 *Device driver head.*

cleared using the background character and attribute specified in the *wn_table* data structure, with a call to *clear_window()*. The function *box()*, places a border, described in the *box_table* data structure, around the perimeter of the window. The *sayat()* function then places the critical error statement within the window. The *sayat()* function is used throughout the user interface to display null terminated strings at specified row/column coordinates, using a specified attribute.

Error Identification

Deciphering the data pertaining to the actual error begins by determining whether the error was a disk or non-disk error. See Figure 10.2 for error information and location. The variable *err_drive* contains the value which was in the *AX* register when the replacement error handler gained control. Bit *15* is checked by *AND*ing *err_drive* with *8000h* to determine if the error was a disk error. If bit *15* is *0*, the error occurred on a disk, and bits *0* through *7* contain a byte value representing the drive number of the failing drive. Drives are numbered A: = 0, B: = 1, C: = 2, etc. The string "*Disk error on drive *:*" is built, and the appropriate drive letter is inserted in place of the asterisk by *AND*ing out the high byte of *err_drive* and adding the value of an ASCII "A" to obtain the drive letter. The completed string is displayed by the *sayat()* function. The failing area of the drive is contained in bits *9* and *10* of *err_drive*. These two bits can be converted to an index into the *area[]* array by shifting *err_drive* right nine bits (bringing bits *9* and *10* into bit position 0 and 1), then *AND*ing the result with three. The string describing the failing area is then displayed using the *sayat()* function. If bit *15* of *err_drive* is non-zero, the error is a non-disk error. In this case, *dev_driver_ptr* contains the address of the Device Header (Figure 10.3) for the failing device. If the device is a character device, the device name is located 10 bytes into the Device Header. The name is left-justified and padded on the right with blanks. To determine if the device is a character device, examine bit *15* of the device attribute word in the device header. If bit *15* is on, the device is a character device. If it is off, the device is a block device. If the failing device is a block device, the string "*UNKNOWN*" is concatenated as the device name, otherwise the device name from the device header is concatenated and the name string is displayed.

Error Descriptions

Now that the user knows what failed, it's time to tell *why* it failed. The array of error descriptions, `my_err_codes[]` (Listing 10.5), may contain any 16 descriptions you like. Whatever messages make sense to your users can be placed in this array. You are not bound to the messages I have chosen. Mine are pretty close to the default DOS messages anyway. The index into the `my_error_codes[]` array is built from the lower 8 bits if the variable `err_code` (contents of the `DI` register when the error handler was entered). The high 8 bits are `AND`ed out, and the result is an error number between 0 and 15 for DOS v3.x, or 0 and 12 for DOS v2.x.

Before invoking the error handler, DOS determined what options the user should be given. These options were coded into three bits of the `AX` register when the error handler received control. Bits `11`, `12`, and `13` of the variable `err_drive` can now be used to determine valid user options. Bits `11`, `12`, and `13` are shifted right into bit positions 0, 1, and 2 then `AND`ed with `00ffh`. The result can now be tested against 2, and 4 to see if retry and/or ignore are valid. The abort option and the fail option will always be available with this replacement error handler. The code beginning immediately after the comment "display only the valid options" determines which options are valid, displays the option message, and fills the `valid_options` array with the scan code of the keystroke assigned to that option. Along the way, the `hilight()` function changes the attribute of the first character in each message displayed to indicate the keystroke tied to that option.

User Responses

Now it's time to get the user's response. All keystrokes stored in the type ahead buffer are discarded with a call to the function `flush_kbd_buffer()`, and the `inkey()` function waits for and returns an unsigned integer representing the next key pressed by the user. The scan code of the key is contained in the high byte, and the ASCII character is in the low byte. Shifting the high byte of the returned keycode into the low byte results in an integer representing only the scan code returned by the keyboard. By dealing with scan codes, it is not necessary to make case insensitive comparisons of ASCII characters. For example, the scan code returned when the letter "Q" is pressed is the same, regardless of the state of the shift key or the caps lock key. When one of the recognized scan codes is returned, the corresponding element of the `valid_options` array is examined to determine if that option is available to the user. If the option is valid, `ret_val` is set to the appropriate value, and the `while` loop terminates. If the scan code is not recognized, or the option is not acceptable, the keystroke is discarded and the `while` loop is executed again. When the `while` loop terminates, the variable, `ret_val`, holds the correct value to return to the replacement error handler. The entry screen is restored with a call to `restore_window()`, and the users response is returned to the replacement error handler.

Clean Up

Listing 10.3 contains the `clean_up_for_exit()` function. This `clean_up_for_exit()` function does nothing more than use the C library function `printf()` to display a message and then wait for a key to be pressed before returning to the replacement handler, which will then terminate the process. Remember that this clean-up routine can do anything you like. Keep it simple. Do only the absolute minimum which will allow your application to gracefully exit. Try not to do anything which could cause a DOS critical error.

To test the replacement handler, the function `main()` (Listing 10.3) installs the replacement error handler and attempts to write to the LPT1 device or to open a file on the A drive. If a command line parameter is present (what it is doesn't matter), the C library function `fputs()` attempts to write the string, "This should go to the printer," to LPT1. If the printer is off-line or out of paper, the replacement error handler will receive control and display the appropriate message. One note of caution here: If the DOS `Mode` command has been used on LPT1, and the `/P` option was specified (infinite retry), DOS won't generate a critical error. The infinite retry option caught me, and it took several minutes of head scratching before I finally figured out what the problem was. If there is no command line parameter present, the C library function `open()` is called to open a file on the A drive. If the drive is empty, or contains an unformatted disk, DOS generates a critical error and the replacement error handler receives control. A make file to build the test driver is provided as Listing 10.7. The assembly language modules "include" the `cmacro.inc` file shown in Listing 10.6.

An important part of the development process which is often overlooked is the handling of error conditions. With the information and code presented here, you now have the necessary tools to trap critical errors and display meaningful messages to the user. Professional applications handle things like critical errors, as well as user-generated exceptions like *Ctrl-C* and *Ctrl-Break*, with grace. As the level of sophistication of the PC user has increased, so has the demand that sophistication of PC applications increase. Taking extra time to insure your program will withstand worst-case scenarios makes it more valuable to the user.

References:

Disk Operating System Version 3.30 Technical Reference, First Edition, International Business Machines Corporation, April 1987.

Hogan, Thom. *The Programmer's PC Sourcebook*, Microsoft Press, Redmond, Washington, 1988.

Rollins, Dan. "DOS Exception Handling," *PC Tech Journal,* April, 1987.

Listing 10.1 `critical.asm`

```
COMMENT @
---------------------------------------------------------
Author:  David Burki
 Contents:  int24()     critical error handler replacement
            install24() routine to handle installation of
                        the replacement error handler.
 Borland TASM version 2.0

 Copyright 1990, David Burki
---------------------------------------------------------
END OF COMMENT @

; -- memory model and language interface
.model large, C

; -- macro include file
include cmacro.inc

; -- data which is declared in the user interface module
.data
comm near dos_function:word
comm near err_drive:word
comm near err_code:word
comm near dev_driver_ptr:dword
;comm near dev_driver_seg:word

.code

public int24
public install24

; -- stack pointer is saved here on entry to error handler
save_sp         dw      0

; -- dword address of the two functions which the
;    replacement error handler will call if non-zero
dialog_func     dd      0
pre_exit        dd      0
```

Listing 10.1 *continued*

```
assume cs:@curseg, ds:nothing, es:nothing
;--------------------------------------------------------
;   INT24()
; This is the actual replacement routine for interrupt
; 24h (critical error handler).
;--------------------------------------------------------
int24 proc far
; -- save the entry stack pointer, any
;    register that might be used & the flags
        mov     cs:save_sp,sp
        apush   ds,es,bx,cx,dx,di,si
        pushf

; -- if a user interface function is installed, pass
;    control to the user interface function
        cmp     cs:word ptr dialog_func+2,0
        jnz     ask_user

; -- otherwise proceed to "fail" the function call
        mov     ax,3
        jmp     short stabilize

; -- address the common data segment and store all the
;    information about the error where the user interface
;    function has access to it, including the function
;    number of the failing function.  then call the user
;    interface routine.
ask_user:
        mov     bx,DGROUP
        mov     ds,bx
assume ds:DGROUP
        mov     err_drive,ax
        mov     err_code,di
        mov     word ptr dev_driver_ptr,si
        mov     word ptr dev_driver_ptr+2,bp
        mov     bp,cs:save_sp
        mov     ax,[bp+6]
        mov     dos_function,ax
        call    cs:[dialog_func]

; -- if the user selected retry or ignore, give control
;    back to DOS.  (al = 0 or 1)
        cmp     ax,1
        jle     back_to_dos

; -- otherwise make a DOS call which will stabilize DOS
;    as recomended in the Tech. Ref.
stabilize:
        push    ax
        mov     ah,30h
        int     21h
        pop     ax
```

Listing 10.1 *continued*

```
; -- if the user requested to quit (abort), jump to the
;    abort section of this code
        cmp     ax,2
        jz      do_abort_routine

do_fail:
; -- otherwise, the user must have requested fail or no
;    user dialog took place
;    restore SP to where it was when handler was
;    first entered, retrieve the function number which
;    failed, and stuff the appropriate return value for
;    a fail into the stack position which will be popped
;    back into AX.
        mov     sp,cs:save_sp
        mov     bp,sp
        mov     bx,[bp+6]
        cmp     bh,0fh
        jl      set_ah
        cmp     bh,36h
        jg      set_error_num
        jne     set_al
        mov     ax,-1
        jmp     short continue_fail
set_al:
        mov     al,-1
        jmp     short continue_fail
set_ah:
        mov     ah,0ffh
        jmp     short continue_fail
set_error_num:
        mov     ax,err_code

continue_fail:
; -- remove ip, cs, flags, and ax pushed when int 24h invoked
        add     sp,8

; -- restore regs to state they were when int 21h dispatcher
;    got control - except ax now has the proper fail
;    indication for the function that failed
        apop    es,ds,bp,di,si,dx,cx,bx

; -- set the carry as indicator that ax contains
;    an error code and return to instruction immediately
;    following the int 21h which failed.
        stc
        ret     2

; -- here is where we clean up the stack and call our
;    clean-up routine if installed, trying to make
;    some kind of graceful exit.
```

Listing 10.1 *continued*

```
do_abort_routine:
; -- restore the entry stack pointer and discard the
;    registers DOS saved before invoking int 24h
        mov     sp,cs:save_sp
        add     sp,30

; -- if clean-up routine is not installed, just quit
        cmp     cs:word ptr pre_exit+2,0
        jz      exit_to_os

; -- otherwise call the clean-up routine
        call    cs:[pre_exit]

exit_to_os:
; -- terminate with errorlevel
        mov     ax,4cffh
        int     21h

back_to_dos:
; -- at this point ax has the appropriate return code
;    0 to ignore or 1 to retry the DOS function which
;    had the error
;    restore all registers saved when the handler was
;    entered, restore SP to its entry value and return
;    to DOS
        popf
        apop    ds,es,bx,cx,dx,di,si
        mov     sp,cs:save_sp
        iret
int24 endp
```

Listing 10.1 *continued*

```
assume cs:@curseg, ds:nothing, es:nothing
;-------------------------------------------------------
;   INSTALL24()
; Installs the replcaement error handler and stores the
; pointers to both the user interface routine and the
; clean-up routine in cs addressable variables for the
; replacement handlers use.
; NOTE:  both pointers are expected to be double word
;        addresses, and either/both may be null.
;-------------------------------------------------------
install24 proc
 USES DS
 ARG  user_interface:CODEPTR, clean_up:CODEPTR
        lds     ax,user_interface
        mov     cs:word ptr dialog_func,ax
        mov     cs:word ptr dialog_func+2,ds
        lds     ax,clean_up
        mov     cs:word ptr pre_exit,ax
        mov     cs:word ptr pre_exit+2,ds
        push    cs
        pop     ds
assume ds:@curseg
        lea     dx,int24
        mov     ax,2524h
        int     21h
        ret
install24 endp

end

; End of File
```

Listing 10.2 user_io.c

```
/*--------------------------------------------------------
  Author:  David Burki

  Copyright 1990, David Burki
--------------------------------------------------------*/
#include <errs.h>   // global defines & data
                    // for user interface
#include <dos.h>

/* --------------------------------------------------------
    USER_INTERFACE()
    This function implements the mainline code for a user
    interface routine to handle a critical error.  It
    chooses the appropriate messages found in errs.h and
    calls direct video write routines to display them.

    NOTE: no DOS function call higher than 0x0c can be
          made within this code.
 --------------------------------------------------------*/
int user_interface()
{
int i,
    start_text_col,
    options_row,
    ret_val,
    done,
    wn_attr,
    hilite_attr;
char temp_str[7];
unsigned valid_options[4],
         useit;

        // find the segment address of video ram since
        // we're doing direct video access & set attribute
  if( monitor() == MONO_DISPLAY )
     {
     wn_attr = MONO_ATTRIBUTE;
     hilite_attr = MONO_KEY_ATTR;
     }
  else
     {
     wn_attr = COLOR_ATTRIBUTE;
     hilite_attr = COLOR_KEY_ATTR;
     }
```

Listing 10.2 *continued*

```
        // find the center of the window
  useit = (wn_table[1]+((wn_table[3]-wn_table[1])/2));

        // start messages 2 cols in from the window edge
  start_text_col = wn_table[1] + 2;

        // zero out valid options array

  memset(valid_options, 0, 2*sizeof(unsigned int));

        // save the area where the window will be
  save_window(wn_table, save_buf);

        // clear the window to background color
  clear_window(wn_table);

        // draw the box border around the window
  box(wn_table, box_table);

        // make the statement
  strcpy(error_str, "A Critical Error has occurred!");
  sayat(6, useit - (strlen(error_str)/2), error_str,
        wn_attr);

// here is where the information about the cause of
// the error is deciphered and presented to the user.

        // was it a disk error (bit 15 is zero)
  useit = err_drive & 0x8000;
  if( useit == 0 )
     {
     strcpy(error_str, "Disk error on drive *:");
     i = strlen(error_str)-2;
     useit = err_drive & 0x00ff;
     error_str[i] = (char)useit + 'A';
     sayat(8, start_text_col, error_str, wn_attr);

     strcpy(error_str, "The failing area is the ");
     useit = (err_drive >> 9) & 0x0003;
     strcat(error_str, area[useit]);
     sayat(9, start_text_col, error_str, wn_attr);
     strcpy(temp_str, "disk.");
     }
```

Listing 10.2 *continued*

```
      // or was it a device error
else
   {
   strcpy(error_str, "The failing device is ");

      // if the device is a block device
   if( *(dev_driver_ptr + 4) & 0x8000 )
      {
      strcat(error_str, "UNKNOWN");
      sayat(8, start_text_col, error_str, wn_attr);
      strcpy(error_str, "Block device systems error.");
      sayat(9, start_text_col, error_str, wn_attr);
      }
      // not block so its a character device
   else
      {
      // use the pointer to the device driver header
      // and bump the pointer to the device name
      dev_driver_ptr += 10;
      i = strlen(error_str);

      // concat the device name
      for(useit = 0; useit < 8 && *dev_driver_ptr != BLANK;
             useit++)
         error_str[i++] = *dev_driver_ptr++;

      // terminate with null and display string
      error_str[i] = '\0';
      sayat(8, start_text_col, error_str, wn_attr);
      strcpy(temp_str, "device");
      }

   }

      // build the error explaination & display it
sayat(10, start_text_col, "The error is:", wn_attr);

      // mask out the high byte
useit = err_code & 0x00ff;

      // and use result as index into array of error codes
strcpy(error_str, my_error_codes[useit]);

      // if it was a write error, concat "disk" or
      // "device" as appropriate
if( useit == 10 )
   strcat(error_str, temp_str);

      // display the message
sayat(11, start_text_col + 4, error_str, wn_attr);

      // init row where valid options start
options_row = B_ROW - 4;
```

Listing 10.2 *continued*

```
// display only the valid options

        // mask out all but the option bits
  useit = err_drive >> 11;
  useit &= 0x0007;

        // check the ignore option
  if( useit & IGNORE_MASK )
     {
     valid_options[IGNORE_OPTION] = SCAN_I;
     sayat(options_row, start_text_col, ignore_string,
           wn_attr);
     hilite(options_row, start_text_col, 1, hilite_attr);
     options_row++;
     }

        // check the retry option
  if( useit & RETRY_MASK )
     {
     valid_options[RETRY_OPTION] = SCAN_R;
     sayat(options_row, start_text_col, retry_string,
           wn_attr);
     hilite(options_row, start_text_col, 1, hilite_attr);
     options_row++;
     }

        // the abort (quit option) is always allowed
  valid_options[QUIT_OPTION] = SCAN_Q;
  sayat(options_row, start_text_col, quit_string,
        wn_attr);
  hilite(options_row, start_text_col, 1, hilite_attr);
  options_row++;

        // with our handler fail is always allowed
  valid_options[FAIL_OPTION] = SCAN_F;
  sayat(options_row, start_text_col, fail_string,
        wn_attr);
  hilite(options_row, start_text_col, 1, hilite_attr);

        // flush the keyboard type ahead buffer
  flush_kbd_buffer();
```

Listing 10.2 *continued*

```
// enter a loop which gets the keyboard scan code of keys
// pressed by the user and won't exit until one of the
// valid option letters has been pressed.

  done = FALSE;

  while( !done )
     {
     done = TRUE;   // assume done until proven otherwise

     switch( (inkey() >> 8) )
        {
        case SCAN_I:
           if( valid_options[IGNORE_OPTION] )
              ret_val = IGNORE_OPTION;
           break;

        case SCAN_R:
           if( valid_options[RETRY_OPTION] )
              ret_val = RETRY_OPTION;
           break;

        case SCAN_Q:
           if( valid_options[QUIT_OPTION] )
              ret_val = QUIT_OPTION;
           break;

        case SCAN_F:
           if( valid_options[FAIL_OPTION] )
              ret_val = FAIL_OPTION;
           break;

        default:    // key wasn't one of the allowed
           done = FALSE;
           break;
        }
     }

        // put back the original display
  restore_window(wn_table, save_buf);

        // return the selected option
  return ret_val;
}

/* End of File */
```

Listing 10.3 `crittest.c`

```
/*------------------------------------------------------
  Author:  David Burki

  Copyright 1990, David Burki
-------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

        // prototype the external functions
extern void monitor(void);
extern int user_interface(void);

        // prototype local functions
int main(int, char *[]);
void clean_up_for_exit(void);

        // define a pointer to a function with no
        // return value
typedef void (*VOID_FPTR)();

        // define a null function pointer
#define NULL_FPTR ((VOID_FPTR) 0)

        // string to be written if no error
char *test_str = "Hello World";
```

Listing 10.3 *continued*

```
main(int argc, char *argv[])
{
int handle;

        // install the replacement error handler
        // either parameter may be NULL_FPTR if desired
  install24( user_interface, clean_up_for_exit );

        // any command line parm & we'll try the printer
  if( argc > 1 )
     fputs("This should go to the printer\n",stdprn);

        // no command line parm & we'll try the disk
  else
     {
     if( (handle = open("a:abc.xyz", O_RDONLY) == -1) )
        {
        printf("\nfailed the open call\n");
        printf("The error is %d\n", errno);
        return 1;
        }
     else
        {
        printf("\nback after open() call without error\n");
        close(handle);
        }
     }
  return 0;
}

void clean_up_for_exit()
{
  printf("Here we are in the clean-up routine.\n\n");
  printf("Any processing necessary to gracefully\n");
  printf("exit your program can be done here.\n");
  printf("Use caution and don't get carried away though.\n");
  printf("Critical errors here could get hairy!\n\n");
  printf("Press any key to exit.\n");
  while( !kbhit() )
     ;
}

/* End of File */
```

Listing 10.4 `scrnstuf.asm`

```
COMMENT @
-----------------------------------------------------------
 Author:  David Burki
 Contents: calc_scrn_addr()
           clear_window()
           box()
           save_window()
           restore_window()
           calc_boxsize()
           hilite()
           monitor()
           sayat()
           inkey()
           flush_kbd_buffer()

 Borland TASM version 2.0

 Copyright 1990, David Burki
-----------------------------------------------------------
END OF COMMENT @

; -- memory model and language interface
.model large, C

; -- macro include file
include cmacro.inc

; equate corresponding to BIOS data segment
ROM     equ     40h

.data
; -- segment address of video RAM -- default = mono
screen_segment  dw      0b000h

.code
        public clear_window
        public box
        public save_window
        public restore_window
        public calc_boxsize
        public hilite
        public monitor
        public sayat
        public inkey
        public flush_kbd_buffer
```

Listing 10.4 *continued*

```
;-------------------------------------------------------------
;   CALC_SCRN_ADDR()
; calculate the byte offset of a given row/column
; coordinate;
; ENTRY:   dh - row
;          dl - column
; RETURNS: di - byte offset of row/column
;          es - segment address currently stored in
;               screen_segment variable.
;-------------------------------------------------------------
calc_scrn_addr proc
        mov     ah,dh
        xor     al,al           ;ax = 256*row
        shr     ax,1            ;    128 * row
        mov     di,ax           ;di =128*row
        shr     ax,1            ;ax =64*row
        shr     ax,1            ;    32*row
        add     di,ax           ;di=128+32*row=160*row
        mov     al,dl           ;al=col
        cbw                     ;ax=col
        shl     ax,1            ;ax=2*col
        add     di,ax           ;di=160*row+2*col
        mov     ax,DGROUP
        mov     es,ax
assume es:DGROUP
        mov     es,es:screen_segment
assume es:nothing
        ret
calc_scrn_addr endp

;-------------------------------------------------------------
; CALC_BOXSIZE()
;  Calculates the number of rows and columns within a
;  window defined by a window table data structure
;  ENTRY:   ds:si pointing to a window table
;  RETURNS: bx containing the number of rows
;           cx containing the number of columns
;-------------------------------------------------------------
calc_boxsize proc
        mov     bl,[si+2]   ;bottom row
        sub     bl,[si]     ;subtract top row
        inc     bl          ;and add 1.
        xor     bh,bh       ;bx now = num rows
        mov     cl,[si+3]   ;right column number into cl
        sub     cl,[si+1]   ;subtract left column number
        inc     cl          ;and add 1
        xor     ch,ch       ;cl has number of columns wide
        ret
calc_boxsize endp
```

Listing 10.4 *continued*

```
;--------------------------------------------------------
; CLEAR_WINDOW( char *window_table )
;
;  Clears the area of the screen defined by the window
;  table data structure.
;  Window table data structure is:
;     {top_row, left_col, bottom_row, right_col,
;      bg_char, attribute}
;--------------------------------------------------------
clear_window proc
 USES ES, DS, SI
 ARG  window_table:DATAPTR
if @datasize
        lds     si,window_table
else
        mov     si,window_table
endif
        call    calc_boxsize   ;returns box dimensions
        mov     dh,[si]
        mov     dl,[si+1]       ;row/col into dx
        call    calc_scrn_addr ;setup es:di to vidio ram
        mov     ah,[si + 5]     ;put attr from table in ah
        mov     al,[si + 4]     ;character to clear with
        cld                     ;make sure dir. flag is fwd
next_col:
        push    cx              ;save number of columns
        push    di              ;save original offset
        rep     stosw           ;put up a row
        pop     di              ;get back number of columns
        pop     cx              ;get back starting offset
        add     di,160          ;next row starting offset
        dec     bl              ;subtract for row just done
        jnz     next_col        ;go do it again
exit:
        ret
clear_window endp
```

Listing 10.4 *continued*

```
;--------------------------------------------------------
; BOX( char *window_table, char *box_table )
;  Draws a rectangular frame defined by top_row, left_col
;  and bottom_row, right_col using the attribute defined
;  by attribute.  The frame characters are defined by
;  box_table.
;  Window table data structure is:
;     {top_row, left_col, bottom_row, right_col,
;      bg_char, attribute}
;  Box table data structure is:
;     {ul_corner_char, ur_corner_char, top_bot_char,
;      sides_char, ll_corner_char, lr_corner_char}
;--------------------------------------------------------
box proc
 USES ES, DS, SI
 ARG  window_table:DATAPTR, box_table:DATAPTR
if @datasize
        lds     si,window_table
else                           ;ptr to window table
        mov     si,window_table
endif
        call    calc_boxsize   ;returns box dimensions
        sub     bl,2           ;and sub 2. bl = num rows
                               ;   with side char
        sub     cl,2           ; to get number of columns
                               ;   between side characters
        mov     dh,[si]
        mov     dl,[si+1]      ;row/col into dx
        call    calc_scrn_addr ;setup es:di to vidio memory
        mov     ah,[si+5]      ;attr for box
        cld                    ; direction flag fwd
pane_top:
        push    di             ;save original offset
if @datasize
        lds     si,box_table
else                           ;ptr to box table
        mov     si,box_table
endif
```

Listing 10.4 *continued*

```
        mov     al,[si]         ;move upper left corner char
        stosw                   ;put it up
        mov     al,[si+2]       ;top & bottom frame char
        push    cx              ;save width
        rep     stosw           ;put up top of frame
        pop     cx              ;get width back
        mov     dx,cx           ;save number of columns
        add     cx,cx           ;and set it for right side
                                ;  column offset
        mov     al,[si+1]       ;move upper right corner char
        stosw                   ;put it up
        pop     di              ;get back original offset
        cmp     bl,0            ;done all rows?
        je      pane_bottom
pane_sides:
        add     di,160          ;get next row offset
        push    di              ;and save it
        mov     al,[si+3]       ;move pane side char
        stosw                   ;put up left side
        add     di,cx           ;find right side offset
        stosw                   ;put up right side pane char
        pop     di              ;get back last row/col offset
        dec     bl              ;subtract for row just done
        jnz     pane_sides      ;do it again
pane_bottom:
        add     di,160          ;get next row/col offset
        mov     al,[si+4]       ;move lower left corner char
        stosw                   ;put it up
        mov     al,[si+2]       ;top & bottom char
        mov     cx,dx           ;get number of cols into cx
        rep     stosw           ;put up bottom of pane
        mov     al,[si+5]       ;move lower left corner char
        stosw                   ;put it up
        ret
box endp

;---------------------------------------------------------
;RESTORE_WINDOW( char *window_table, char *save_area )
;  Restores the retangular area of the screen saved in
;  the save area buffer by a previous call to save_window.
;  See clear_window() for the definition of the window
;  table data structure.
;---------------------------------------------------------
restore_window proc
 USES ES, DI, DS, SI
 ARG  wn_table:DATAPTR, save_area:DATAPTR
if @datasize
        lds     si,wn_table
else                            ;pointer to wn table
        mov     si,wn_table
endif
```

Listing 10.4 *continued*

```
        call    calc_boxsize    ;get # rows & # columns
        mov     dh,[si]         ;row into dh
        mov     dl,[si+1]       ;col into dl
        call    calc_scrn_addr  ;set up for screen
if @datasize
        lds     si,save_area
else                            ;pointer to save area
        mov     si,save_area
endif
        cld                     ;direction flag is fwd
r_next_row:
        push    cx              ;save number of columns
        push    di              ;save original offset
        rep     movsw           ;put back a row
        pop     di              ;get back number of columns
        pop     cx              ;get back last starting offset
        add     di,160          ;find next row starting offset
        dec     bl              ;subtract for row just done
        jnz     r_next_row      ;go do it again
        ret
restore_window endp

;-------------------------------------------------------------
;SAVE_WINDOW( char *window_table, char *save_area )
;  Saves the retangular area of the screen defined by the
;  window table data structure into the buffer pointed to
;  by the save area pointer.  See clear_window() for the
;  definition of the window table data structure.
;-------------------------------------------------------------
save_window proc
 USES ES, DI, DS, SI
 ARG  wn_table:DATAPTR, save_area:DATAPTR
if @datasize
        lds     si,wn_table
else                            ;pointer to wn table
        mov     si,wn_table
endif
        call    calc_boxsize    ;box # rows & # columns
        mov     dh,[si]
        mov     dl,[si+1]       ;row/col into dx
        call    calc_scrn_addr  ;set up for screen
        push    es              ;save screen seg & offset
        push    di
if @datasize
        les     di,save_area
else                            ;pointer to save buffer
        mov     di,save_area
endif
```

Listing 10.4 *continued*

```
        pop     si              ;swap source
        pop     ds              ;  and destination
        cld                     ;direction flag is fwd
s_next_row:
        push    cx              ;save number of columns
        push    si              ;save original offset
        rep     movsw           ;get down a row
        pop     si              ;get back number of columns
        pop     cx              ;get back last starting offset
        add     si,160          ;find next row starting offset
        dec     bl              ;subtract for row just done
        jnz     s_next_row      ;go do it again
        ret
save_window endp

;-------------------------------------------------------
;SAY_AT( char row, char col, char *string, char attr )
;  Display the null terminated string at row/column
;  coordinates using the specified attribute.
;  Row and column are based on screen coordinates where
;  (0,0) is the upper left corner of the display.  No
;  check is made for line wrapping or snow control.
;-------------------------------------------------------
sayat proc
 USES DS,SI,ES,DI,DX
 ARG  row:WORD, col:WORD, string:DATAPTR, attr:WORD
        mov     dh,byte ptr row  ;get the row
        mov     dl,byte ptr col  ; and column into dx
if @datasize
        lds     si,string
else                             ;pointer to string
        mov     si,string
endif
        call    calc_scrn_addr   ;sets es:di screen ram
        mov     ah,byte ptr attr ;get the attribute
        cld                      ;direction flag to forward
get_a_char:
        lodsb                    ;char into al
        or      al,al            ;is it zero
        jz      all_done         ;it is, so leave
        stosw                    ;put it on the screen
        jmp     get_a_char       ;go get another
all_done:
        ret
sayat endp
```

Listing 10.4 *continued*

```
;--------------------------------------------------------
;  HILITE( char row, char col, int len, char attr )
;    Change len number of attribute bytes in vidio memory
;    to attr starting at row/column.
;    Does not affect the character byte.
;--------------------------------------------------------
public hilite
hilite proc
 USES ES,DI
 ARG  row:WORD, col:WORD, len:WORD, attr:WORD
        mov     dh,byte ptr row  ;get the row
        mov     dl,byte ptr col  ; and column into dx
        call    calc_scrn_addr   ;set es:di to row/col address
        inc     di               ;di points to the char...
                                 ;      make it point to attr
        mov     cx,len           ;how many to change
        mov     al,byte ptr attr ;get the new attribute
        cld                      ;direction forward
around:
        stosb                    ;change one attribute byte
        inc     di               ;move to next attribute byte
        loop    around           ;till cx = zero
        ret
hilite endp

;--------------------------------------------------------
; MONITOR()
;  Determines the active display device.
;  Sets screen_segment to 0b000h if MDA/HERC is active
;  or 0b800h for CGA/EGA/VGA.
;
;  RETURNS: vidio segment in ax
;--------------------------------------------------------
monitor proc
 USES DS, ES
        mov     ax,DGROUP
        mov     ds,ax                 ;address data segment
        mov     ax,ROM
        mov     es,ax                 ;BIOS data segment
        cmp     es:byte ptr[49h],7    ;check crt mode byte
        jz      adios                 ;default to mono
        mov     screen_segment,0b800h ;non-mono segment
adios:
        mov     ax,screen_segment     ;save segment
        ret
monitor endp
```

Listing 10.4 *continued*

```
;-----------------------------------------------------------
; INKEY()
;  Call BIOS to get the next available key in the keyboard
;  buffer.  Wait for a keypress if buffer empty.
;
;  RETURNS: ax contains scan and ascii value of key.
;-----------------------------------------------------------
inkey  proc
        mov     ah,0        ;ask BIOS for a key
        int     16h         ;& wait if one not ready
        ret                 ;ax has the scan/ascii code of key
inkey endp

;-----------------------------------------------------------
; FLUSH_KBD_BUFFER()
;  Remove all keystrokes waiting in the keyboard type
;  ahead buffer.
;-----------------------------------------------------------
flush_kbd_buffer proc
        mov     ah,1
        int     16h                 ;BIOS check of kbd buffer
        jz      buf_is_empty        ;zflag says it's empty
        call    inkey               ;remove waiting key
        jmp     flush_kbd_buffer ;go back & look again
buf_is_empty:
        ret
flush_kbd_buffer endp

end

; End of File
```

Listing 10.5 `errs.h`

```
/*-------------------------------------------------------
  Author:  David Burki

  Copyright 1990, David Burki
------------------------------------------------------*/

//-------------------------------------------------------
// -- global data to construct error messages from
//-------------------------------------------------------

        // text to insert in message based on bit
        // 0 of register AH
char *operation[] =
   {
   "read",
   "write"
   };

        // text to insert in message based on bits
        // 1 and 2 of register AX
char *area[] =
   { "DOS area",
     "File Allocation Table",
     "Disk directory",
     "Disk data"
   };

        // Array of error messages based on bits
        // 0 - 7 of DI register
char *my_error_codes[] =
   {
   "Attempted to write to a write protected disk.",
   "Unknown unit.",
   "Drive is not ready. Check drive door closed.",
   "Unknown command.",
   "Data error (CRC failed) on drive.",
   "Bad request structure length.",
   "Seek error (r/w head position request failed) on drive",
   "Unkown disk media type.",
   "Disk sector not found.",
   "Printer is out of paper",
   "Error writing to ",
   "Error reading disk.",
   "General failure. Check disk is formatted.",
   "Disk error reading drive",
   "UNDEFINED (reserved by DOS) error.",
   "UNDEFINED (reserved by DOS) error.",
   "Invalid disk change."   // DOS 3.x and higher only
   };
```

Listing 10.5 *continued*

```
// ----------------------------------------------------
// -- #defines and global data for the user interface --
// ----------------------------------------------------

        // scan codes of 'I', 'R', 'Q', & 'F'
#define SCAN_I 0x17
#define SCAN_R 0x13
#define SCAN_Q 0x10
#define SCAN_F 0x21

#define IGNORE_OPTION 0
#define RETRY_OPTION  1
#define QUIT_OPTION   2
#define FAIL_OPTION   3

#define IGNORE_MASK   4
#define RETRY_MASK    2


#define TRUE  1
#define FALSE 0
#define BLANK ' '
#define MONO_DISPLAY 0xb000

        // these defines control the location and size
        // of the error message window
#define T_ROW  4        // top row
#define L_COL  10       // left column
#define B_ROW  17       // bottom row
#define R_COL  69       // right column

        // defines for the number of rows and columns
        // in the message window.
#define NUM_ROWS  B_ROW - T_ROW + 1
#define NUM_COLS  R_COL - L_COL + 1

        // based on the size of the window, this define
        // gives the size of the buffer necessary to save
        // all character & attribute bytes in the window
        // Note parentheses around num_cols - required to
        // correctly evaluate since num_cols is expanded to
        // the definition above during compile.
#define BYTES_IN_WIND  ((NUM_COLS) * 2) * NUM_ROWS

        // color attribute of the message window
#define COLOR_ATTRIBUTE 0x4f       // white on red
#define COLOR_KEY_ATTR  0x4a       // green on red
#define MONO_ATTRIBUTE  0x70       // black on white
#define MONO_KEY_ATTR   0x07       // white on black
```

Listing 10.5 *continued*

```
        // user interface prompt strings
char *ignore_string =
 "Ignore the error & continue program. (NOT reccomended)";
char *retry_string =
 "Retry the operation after correcting the problem.";
char *quit_string =
 "Quit the program & return to the operating system (DOS).";
char *fail_string =
 "Fail the operation, returning failure to the program.";

        // simple data structure defining a window
char wn_table[] = { T_ROW, L_COL, B_ROW, R_COL, BLANK,
                    COLOR_ATTRIBUTE };

        // simple data structure defining the border
        // characters for a window
char box_table[] = { 213, 184, 205, 179, 212, 190 };

        // buffer where the display will be saved
char save_buf[BYTES_IN_WIND];

        // these are the locations where the
        // replacement error handler will store
        // information about the error
unsigned int err_drive,      // drive number if disk err
             err_code,       // DOS error number
             dos_function;   // DOS function number
char far *dev_driver_ptr;    // offset & segment of device
                             // driver if device failed

        // this is the buffer where error strings will be
        // constructed.  may be dynamically allocated as
        // long as the allocation is done BEFORE an error
        // happens.
char error_str[81];

        // prototype the external functions
extern void save_window(char *, char *),
            restore_window(char *, char *),
            sayat(int, int, char *, unsigned),
            hilite(int, int, int, unsigned),
            clear_window(char *),
            box(char *, char *);
            flush_kbd_buffer(void);
extern unsigned inkey(void),
                monitor();

/* End of File */
```

Listing 10.6 `cmacro.inc`

```
;;--------------------------------------------------------
;; Author: David Burki
;; Contents:  apush
;;            apop
;;
;; Macros to push and pop a list of registers.  Note that
;; the apop macro takes its parameters in the same order
;; as the apush.  It pops the stack in reverse order of
;; the parameter list.
;;--------------------------------------------------------

.xlist
.xcref

;; push list of registers
APUSH   MACRO   A,B,C,D,E,F,G,H,I
        IRP X,<A,B,C,D,E,F,G,H,I>
        IFNB <X>
        PUSH    X
        ENDIF
        ENDM
        ENDM

;; pop list of registers
APOP    MACRO   A,B,C,D,E,F,G,H,I
        IRP X,<I,H,G,F,E,D,C,B,A>
        IFNB <X>
        POP     X
        ENDIF
        ENDM
        ENDM

.list
.cref

; End of File
```

Listing 10.7 *Make file for* `crittest.exe`

```
##-------------------------------------------------------
##              LISTING 7
## Make file for crittest.exe
##
##  This make file will create an executable to test
##  the critical error handler.
##-------------------------------------------------------

##
##      makefile  -  Borland C makefile for crittest.exe
##

EXEFILE = crittest.exe
BC_LIB_DIR  = c:\borlandc\lib           # Borland C libraries
BC_INC_DIR  = c:\borlandc\include       # Borland C includes
CR_INC_DIR  = .\                        # where errs.h is found

!ifdef DEBUG
DEBUG_SW = -v
!else
DEBUG_SW =
!endif

CC   = bcc -ml -c -w-par $(DEBUG_SW) -I$(CR_INC_DIR)
TASM = tasm  /MX /ZI /O
LINK = tlink $(DEBUG_SW) /L$(BC_LIB_DIR).


crittest.exe:   crittest.obj user_io.obj critical.obj \
                scrnstuf.obj
        $(LINK) c0l crittest user_io critical scrnstuf, \
                crittest.exe, nul, emu.lib mathl.lib cl.lib;

crittest.obj:   crittest.c crit.h
        $(CC) crittest.c

user_io.obj:    user_io.c errs.h crit.h
        $(CC) user_io.c

critical.obj:   critical.asm cmacro.inc
        $(TASM) critical;

scrnstuf.obj:   scrnstuf.asm cmacro.inc
        $(TASM)  scrnstuf;
```

Chapter 11

Ctrl-C *and* Ctrl-Break *Handling under DOS*

David L. Burki

Over the past few years there have been many discussions about preventing *Ctrl-C* and *Ctrl-Break* from abnormally terminating an application program. These "solutions" have all dealt with replacing the `23H` and `1bH` interrupt vectors. While it is true that replacing these vectors with your own handler can prevent the user from terminating the program by pressing *Ctrl-C* or *Ctrl-Break*, what do you do about the annoying `^C` that DOS puts on the screen? When you have spent time and effort designing the input screen for the user, who wants the `^C` to destroy its appearance? Who wants that `^C` on the screen anyway?

One method of preventing the `^C` from appearing on the screen might be to locate the hardware cursor at some non-displayed location like screenrow 26. That strategy works great until you want to display the hardware cursor or use a DOS screen output function. The following discussion and code demonstrate a method which not only prevents unwanted program termination by *Ctrl-C* and *Ctrl-Break* but, more importantly, prevents MS-DOS from knowing these keys have been pressed. What DOS doesn't know can't show up on your input screen.

The method presented here has been developed and tested on IBM PC- and AT-compatible machines using DOS versions from 2.1 to 4.0. The `INT 16H` handler should be very portable between generic MS-DOS machines. The only prerequisite for this handler to work is that the PC's BIOS must function in a manner similar to that of the IBM PC.

First a little background about how DOS handles *Ctrl-C/Break* checking. During certain functions of `INT 21H`, the internal DOS housekeeping code checks the next available key in the keyboard buffer. Using the BIOS `INT 16H` function `1`, DOS examines the next available key in the keyboard buffer. If the next available key code is `2e03H` (the scan and ASCII code for *Ctrl-C*) or the DOS *Ctrl-Break* flag is `3` (more on the flag in a minute), DOS puts the `^C` on the screen, flushes the keyboard buffer, switches to the user stack, clears the carry flag, and executes `INT 23H`. Among other things, the default `INT 23H` handler flushes the file buffers using function `0dh` of the `INT 21H` handler. It then sets the carry flag and returns with a `RET 2`. When the housekeeping code regains control, it uses the carry flag to determine if it should terminate the process (function `4cH` of `INT 21h`) or continue with the execution of the interrupted function. If your application keeps the type ahead buffer empty using BIOS function calls, DOS may never get a chance to see a *Ctrl-C*. A user might still be able to sneak one in, especially if your application makes DOS function calls to write to the screen.

Ctrl-Break is recognized within the keyboard hardware interrupt (BIOS `INT 9`). When `INT 9` detects the *Ctrl-Break* key combination, code within `INT 9` first flushes the keyboard buffer and then executes an `INT 1bH`. By default the code pointed to by the `INT 1bH` vector contains only two instructions:

```
MOV  CS:[FLAG],3
IRET
```

The first instruction sets the internal DOS flag, mentioned earlier, to indicate *Ctrl-Break* has been pressed. The `IRET` returns to the BIOS keyboard interrupt. This is the flag that DOS examines during its housekeeping code to determine if *Ctrl-Break* has been pressed. If this internal DOS flag equals `3`, the same code which executes when *Ctrl-C* is detected is executed. The `^C` is displayed at the current cursor location, and `INT 23H` is executed. A quick session with `Debug` shows that if you set this flag byte to `3`, DOS places the `^C` on the screen at the current cursor location and executes an `INT 23H`. Since it is called from the keyboard hardware interrupt, the DOS `INT 1bH` handler MUST be replaced in order to prevent *Ctrl-Break* from displaying `^C` on the screen. Flagging *Ctrl-Break* is an asynchronous event. Therefore, *Ctrl-Break* may interrupt a program's execution when *Ctrl-C* won't. In order for *Ctrl-C* to be recognized, the *Ctrl-C* key code has to be at the head of the keyboard buffer when the DOS housekeeping process is executing. *Ctrl-Break*, on the other hand, immediately sets the internal DOS break flag, which will be checked the next time the DOS housekeeping process is executed.

Following is the pseudocode for the section of `INT 21H` that does the checking for *Ctrl-C* and *Ctrl-Break*. `FLAG` is the DOS internal variable that `INT 1bH` sets to `3`, and `NEXT_KEY` is the value returned in `AX` by `INT 16H` function `1`.

```
if FLAG equals 3 OR NEXT_KEY equals CTRL_C then
    print "^C"
    flush keyboard buffer
    switch to user stack
    clear carry flag
    execute INT 23h
    change back to DOS stack
    if carry set then
        terminate program
    else
        continue interrupted DOS function
    endif
endif
```

In the BIOS System Data Area there is a byte at `40:71H` called the `Break Flag`. According to the comment in the listing, this byte flag will have bit `7` set by the BIOS `INT 9` handler when *Ctrl-Break* has been pressed. The BIOS listing for `INT 9` clearly shows bit `7` being set. A session with `Debug` shows that once set, this bit is never cleared. Running Periscope's debugger with a break on memory and tracing interrupts shows `40:71H` is never accessed after being set by `INT 9` through the execution of `INT 23H`. This BIOS `Break Flag` is ignored by DOS. Once set by `INT 9`, it stays set.

Using the routines shown here, your application can spawn a child process, and the *Ctrl-C* and *Ctrl-Break* handlers will remain in effect. Traps are placed in `INT 16h` and `INT 1bH`, so when a child process is spawned the DOS loader does not alter them. Even using the `/p` option to invoke a new `COMMAND.COM` shell leaves the `1bH` and `16H` vectors (and the code they point to) intact. Replacing the `INT 23h` vector is NOT necessary in order to trap *Ctrl-C*. Since DOS won't ever know *Ctrl-C* has been pressed, it has no reason to execute an `INT 23H`.

The routines shown in Listing 11.1 set a flag when *Ctrl-C* or *Ctrl-Break* is detected. An enhanced version of this code might simply ignore a *Ctrl-Break* and return a unique key code for *Ctrl-C*. By returning a unique key code, *Ctrl-C* becomes a valid key combination for your application. Notice that there is a hook in the code for `INT 23H`. This is because there are two (at least) undocumented ways to get the *Ctrl-C* code in DOS to execute. These two key combinations are *Ctrl-2* (top row of the keyboard) and *Alt-3* (on the numeric keypad). By not trapping these key combinations, you can leave yourself a "back door" into the DOS *Ctrl-C* routine. The `INT 23H` replacement routine insures that the interrupt vectors captured during installation are reset to the original values before passing control to the original `INT 23H` code. That way, when your application terminates, there will be a valid `INT 16H` vector for the operating system to use.

The idea behind trapping *Ctrl-C* using the BIOS `INT 16H` is fairly simple: replace the `INT 16H` vector with a pointer to a routine that will make calls to the BIOS but will examine the return values and filter any *Ctrl-C* key codes before returning to the caller. The filter can throw away the unwanted keystrokes (as shown in Listing 11.1) or "fix up" the key code and allow the *Ctrl-C* key combination to be a valid keystroke for the application.

The application program (Listing 11.2, `test16.c`) passes a far pointer to an integer variable as a parameter to the `capture` function (Listing 11.1). This address is stored in a code segment variable called `flag`. Later, when *Ctrl-Break* or *Ctrl-C* are detected, the address of this `flag` is used by the `set_flag` function to set the flag to *1*, alerting the application program that one of these keys has been pressed. The capture function then calls the `install_vectors` function, which saves the original and then installs the replacement vectors for interrupts `16H`, `1bH`, and `23H`. The last thing capture does is set the replaced flag to `TRUE`, indicating that the `orig_16h`, `orig_1bh`, and `orig_23h` doublewords contain valid addresses. The release function will look at this flag before attempting to un-install the replacements.

The `sixteen_handler` function does the hard work. The first six lines check the function request to see if the replacement routine needs to be involved. Any function not requesting a key from the buffer is passed on to the original interrupt. In order to support the enhanced keyboard function calls, the original function value is saved in a code segment variable and the function value in `AH` is `AND`ed with `0efH (1110111b)`. The result is to map the extended keyboard function calls `10H`, `11H`, and `12H` to the original keyboard functions `0`, `1`, and `2`. If the mapped function request is `2` or higher, the call is passed on to the original interrupt. In this case, a far jump is all that is necessary to pass control to the original `INT 16H`. The `CS`, `IP`, and `FLAG` registers of the caller are at the top of the stack where the original `INT 16H` instruction put them. The `IRET` that the BIOS `INT 16H` does will return to the original caller.

Once it has been determined that the function is in range, the next step is to determine if the requested function is a "get key" (function `0`) or an "is key ready" (function `1`) call. There are two distinct blocks within `sixteen_handler` to take care of each function request. For "get key," the code begins at the label `get_key_call`, and for "is key ready" the code begins at the label `keyready_call`. Once the requested function is determined, a jump is made to the appropriate block label. After restoring the original function value from `cs:save_funct`, an interrupt is simulated by pushing the flags and calling the original `INT 16H` address. By simulating an interrupt, a call to the saved address of the original `INT 16H` can be made and control will be returned to this code when the called code completes execution.

In the case of a `keyready` request, on return from the BIOS, the `zero` flag is first tested. If the `zero` flag is set, no key is waiting and it is safe to return to the caller. If the `zero` flag is clear, the first thing to do is see if the value returned by the BIOS is the key code for *Ctrl-C*. If it is not, the CMP instruction used to test for *Ctrl-C* key code will have cleared the `zero` flag, indicating a key is available just as the BIOS would have. A `RET 2` is used to return to the caller, leaving the `FLAG` register intact. If the returned key code is `2e03H` (*Ctrl-C*), the applications break flag is set by calling the `set_flag` function, `AH` is set to `0`, and a simulated interrupt to the original `INT 16H` is done to remove the `2e03H` key code from the keyboard buffer. Once that is accomplished, a jump to the `keyready_call` label is made to see if anything is still available in the buffer. When the BIOS returns a non-*Ctrl-C* key code or the `zero` flag is set indicating no key is in the buffer, the `zero` flag and `AX` are left intact and control is returned to the caller using a `RET 2`.

The "get key" (function `0`) is more straightforward. As in the "is key ready" case, the original function value is restored from `cs:save_funct`. An interrupt to the BIOS is simulated by pushing the flags and calling the saved address of the original `INT 16H`. Since the BIOS doesn't return until a key is available in the keyboard buffer, the only thing for `sixteen_handler` to do is check the returned key code for `2e03H`. If the key code isn't *Ctrl-C*, an `IRET` is executed, returning to the caller. If *Ctrl-C* is present, and since a "get key" function doesn't return to the caller until a key is ready, the code branches back to the `get_key_call` label. The result is that the caller (whether it is the application or DOS) never sees the *Ctrl-C* key code.

The `release` function (Listing 11.1) reverses the installation procedure. The `replaced` flag is set to `FALSE` so a subsequent call to `release` (or a call that isn't preceded by a `capture`) won't attempt to restore the vectors. The application program can capture and release *Ctrl-C* and *Ctrl-Break* as often as necessary during the execution.

To prevent MS-DOS from displaying `^C` on the screen when *Ctrl-Break* is pressed, replace the `INT 1bH` vector with an IRET instruction and DOS won't know it happened. *Ctrl-C* on the other hand involves hiding from DOS the *Ctrl-C* (`2e03h`) returned by the BIOS. There are advantages to filtering the return values of `INT 16H` as a means of controlling *Ctrl-C*. As demonstrated here, the `^C` will not show up on the screen, a flag can be set for a high level language to examine, or with a little more effort, *Ctrl-C* can become a valid key combination for an application. You have control!

Listing 11.1 ctrl_c.asm

```
; title Ctrl-C handler   -   Ctrl-Break handler

COMMENT @
  ----------------------------------------------------

  Microsoft MASM 5.0 or Borland TASM
  Standard C calling conventions

  Author:  David Burki
  ----------------------------------------------------

               ----- SUMMARY  -----

 Traps Ctrl-C keystrokes utilizing a "back-end" filtering process
 attached to the BIOS keyboard I/O routine (INT 16h) and traps
 Ctrl-Break keystrokes by replacing INT 1bh. INT 23h is also be
 replaced in order to trap the undocumented Ctrl-2 Alt-3 key
 combinations.

 There are two callable entry points to this code:
    _capture()
        Installs the new interrupt vectors and stores the address
        of the calling programs variable (an int) in the code
        segment addressable variable "flag". Sets the local
        cs referenced variable "replaced" to true to prevent
        _release() from trying to restore vectors from garbage.
    _release()
        Checks the cs referenced "replaced" flag to avoid restoring
        vectors never saved. Restores the replaced vectors to their
        original values once _capture() has been called, this procedure
        MUST be executed before the application terminates, or INT 16h
        & 1bh will be left pointing to random code and a reboot will
        be necessary. The application may capture and release as
        many times as necessary during it's execution.

    Sample call from C:
        int b_flag = 0;
        main()
          {        /* NOTE: b_flag cast as a far ptr */
          capture( (far *)&b_flag );
          .
          .
          release();
          }
  ----------------------------------------------------
END COMMENT @
```

Listing 11.1 *continued*

```
;match the model directive with the application model
.model small

; -------------------------------------------------------
;  MACROs and EQUATES
; -------------------------------------------------------
sim_int macro  num
    pushf
    call    cs:orig_&num
endm

TRUE    equu     1
FALSE   equ     0

;       large model flag parameter equates
if @CODESIZE
   FLG_OFF equ  [bp+6]
   FLG_SEG equ  [bp+8]
;       small model flag parameter equates
else
   FLG_OFF equ  [bp+4]
   FLG_SEG equ  [bp+6]
endif

.code

; ------------------------------------------------------------------
;The replaced address and the flag pointer are in the code segment
;so they will be accessable to the interrupt replacement code.
; ------------------------------------------------------------------

; far address of application program's Ctrl-C/Break detected flag
flag       dd    0

orig_16h  dd    0           ;far addresses of
orig_1bh  dd    0           ;   the original
orig_23h  dd    0           ;       vectors

; flag to let capture() & release() know there are
; valid addresses in org_??h
replaced  db    FALSE

assume ds:@curseg
```

Listing 11.1 *continued*

```
; -----------------------------------------------------
;the following "installation" procedure is a near call
;even in the large model environment
; -----------------------------------------------------
install_vectors proc near

;get current int 16h vector
    mov     ax,3516h
    int     21h
    mov     word ptr orig_16h,bx

;save the vector we found
    mov     word ptr orig_16h+2,es

;get address of new handler
    lea     dx,sixteen_handler

;set the vector to point to our routine
    mov     ax,2516h
    int     21h

; -------------------------------------------------------
;replacement of 1bh is mandatory if you want to prevent
;int 1bh (Ctrl-Break) from setting the flag DOS looks at
; -------------------------------------------------------

;get current int 1bh vector
    mov     ax,351bh
    int     21h
    mov     word ptr orig_1bh,bx

;save the vector we found
    mov     word ptr orig_1bh+2,es

;get address of new handler
    lea     dx,int1b_handler

;set the vector to point
;to our routine
    mov     ax,251bh
    int     21h
```

Listing 11.1 *continued*

```
; -------------------------------------------------------------
;replacement of 23h is *not* necessary to trap Ctrl-C or Ctrl-Break, however,
;if you want to break from your code using Ctrl-2 or Alt-3, int 23h is
;where we'll make a call to release() so the installed interrupt handlers
;will be un-installed, thus preventing a system hang on return to DOS.
; -------------------------------------------------------------

;get current int 23h vector
    mov     ax,3523h
    int     21h
    mov     word ptr orig_23h,bx

;save the vector we found
    mov     word ptr orig_23h+2,es

;get address of new handlers
    lea     dx,int23_handler

;set the vector to point to our routine
    mov     ax,2523h
    int     21h
    ret
install_vectors endp

; ---------------------------------------------------------------
;_capture uses the ".model" memory model near/far proc
;  NOTE: The parameter "flag" is ALWAYS passed as a far pointer.
; ---------------------------------------------------------------
public  _capture
_capture proc

;establish the stack frame
    push    bp
    mov     bp,sp

;save the application programs ds & es registers
    push    ds
    push    es

;make ds point to the code segment for vector swaps
    push    cs
    pop     ds

;check if already installed
    cmp     replaced,TRUE
    jz      capture_exit

;get the offset and segment of application "break_flag"
    mov     ax,word ptr FLG_OFF
    mov     word ptr flag,ax
    mov     ax,word ptr FLG_SEG
    mov     word ptr flag+2,ax
```

Listing 11.1 *continued*

```
;install the replacements
; NOTE:  near overides for when large model
    call    near ptr install_vectors

;flag that things have changed
    mov     byte ptr replaced,TRUE

;restore registers and stack frame
capture_exit:
    pop     es
    pop     ds
    pop     bp
    ret
_capture endp

assume cs:@curseg, ds:nothing, es:nothing

; -----------------------------------------------------------------------
;_release uses the default memory model near/far proc restores
;interrupt 16h, 1bh & 23h to what they were before capture changed them
; -----------------------------------------------------------------------
public _release
_release proc

;save regs used locally
    push    ds
    push    dx

;save the flags in case this routine has been called
;by the int 23h handler
    pushf

;check that _capture() has installed the handlers
    cmp     cs:replaced,TRUE
    jnz     release_exit

;ds:dx gets the address of the saved original interrupt 16h vector
    lds     dx,cs:orig_16h

;reset the int 16h vector
    mov     ax,2516h
    int     21h

;ds:dx gets the address of the saved original interrupt 1bh vector
    lds     dx,cs:orig_1bh

;reset the int 1bh vector
    mov     ax,251bh
    int     21h
```

Listing 11.1 *continued*

```
;ds:dx gets the address of the saved original interrupt 23h vector
    lds     dx,cs:orig_23h

;reset the int 23h vector
    mov     ax,2523h
    int     21h

;indicate that vectore are no longer replaced
    mov     cs:replaced, FALSE

release_exit:
;restore flags, dx & ds
    popf
    pop     dx
    pop     ds
    ret
_release endp

assume cs:@curseg, ds:nothing, es:nothing

;place to store the int 16h function parameter
;re-entrancy is not a problem
save_funct      db      ?

; ------------------------------------------------------------------
;sixteen_handler is a far proc regardless of the memory model specified
;in the ".model" directive since it is an interrupt replacement routine
; ------------------------------------------------------------------
sixteen_handler proc far

;save the function value
    mov     cs:save_funct,ah

;convert to the non-extended numbers
    and     ah,11101111b

;is it a shift status request
    cmp     ah,2

;lower than shift status request, we'll take care of it
    jb      not_shift_status_req

;put back the callers function, pass it to the BIOS &
;   don't come back here
    mov     ah,cs:save_funct
    jmp     cs:orig_16h

not_shift_status_req:
;if it is it a "is_keyready" call handle it in the
;keyready_call block of code
    cmp     ah,1
    jz      keyready_call
```

Listing 11.1 *continued*

```
;must be a "get_key" request
get_key_call:

;restore the callers original function value
    mov     ah,cs:save_funct

;simulate an interrupt
    sim_int 16h

;did the BIOS return the Ctrl-C keycode
    cmp     ax,2e03h

;no, so we can return to caller
    jnz     iret_back

;the BIOS returned a Ctrl-C keycode, so
;set the flag in the application program
    call    near ptr set_flag

;the Ctrl-C key is thrown away so go back and get another key
    jmp     get_key_call

keyready_call:
;restore the callers original function value
    mov     ah,cs:save_funct

;simulate an interrupt
    sim_int 16h

;if the zero flag is set (by the BIOS), the keyboard
;buffer is empty - ok to return to caller
    jz      ok_to_go_back

;compare the key at the head of the keyboard buffer with the Ctrl-C
;keycode. This compare will leave the Z flag indicating a key is available.
    cmp     ax,2e03h
    jnz     ok_to_go_back

;key was Ctrl-C, set the application program flag
    call    near ptr set_flag

;remove Ctrl-C keycode from the keyboard buffer
    mov     ah,0
    sim_int 16h

;loop back to see if a non Ctrl-C key is ready
    jmp     keyready_call

ok_to_go_back:
;throw away flags of our caller and return
    ret     2
```

Listing 11.1 *continued*

```
iret_back:
;restore callers flags on return
    iret
sixteen_handler endp

; -----------------------------------------------------
;int1b_handler is a far proc regardless of memory model
;   set the application program flag and return
; -----------------------------------------------------
int1b_handler proc far
    call    near ptr set_flag
    iret
int1b_handler endp

; ----------------------------------------------------------------
;int23_handler is a far proc regardless of memory model used here
;  to allow Ctrl-2 or Alt-3 to "break" the program execution
;
;restore the original vectors and execute the original
;Ctrl-C interrupt handler
; ----------------------------------------------------------------
int23_handler proc far
    call    _release
    jmp     cs:orig_23h
int23_handler endp

; -----------------------------------------------------
;set_flag is a near procedure regardless of memory model
;
; Use the address passed to _capture() and set the
; integer refrenced to one.
; -----------------------------------------------------
set_flag proc near
    push    ds
    push    si

;get the address of the application break flag
    lds     si,cs:flag

;set the flag to one
    mov     word ptr [si],1
    pop     si
    pop     ds
    ret
set_flag endp

end

; End of File
```

Listing 11.2 test16.c

```
/* ----------------------------------------------------------------
        TEST16.C
            A driver used to validate the Ctrl-C/Break routines
            in ctrl_c.asm. The C standard library function calls
            use DOS Int 0x21 to obtain keys from the keyboard
            buffer and to display characters. This gives DOS
            plenty of opportunity to execute the Ctrl-C/Break
            checking code within interrupt 0x21.

        Commands to produce an executable from the code as
        shown in the listings:
            masm ctrl_c;
            cl -AL -c test16.c
            link test16 ctrl_c;

        Author:  D. Burki
   ---------------------------------------------------------------- */

#include <stdio.h>
#include <conio.h>  /* borlandc */

/* here we define the key which will allow the input loop to
   terminate. This key must NOT be a cursor movement or function
   key. getch() must return non zero on the initial call when
   this key is pressed in order to exit the do ... while() loop.  */
                                /* exit when enter pressed*/
#define EXIT_KEY 0x0d
                                /* key name for inform()  */
#define EXIT_KEY_NAME "ENTER"

int break_flag = 0;             /* flag set if Ctrl-C or  */
                                /* Ctrl-Break is detected */

void main(void);
void inform(void);

extern void capture(int far *);
extern void release(void);

/*                -----  MAIN  -----
   ------------------------------------------------- */
void main( void )
{
unsigned int ch;
                                /* install the traps      */
  capture((int far *)&break_flag);
  inform();
```

Listing 11.2 *continued*

```
  do                        /* until EXIT KEY pressed */
    {
    printf("If you press Ctrl-C/Break, you must also\n");
    printf("press another key. capture() will eat the \n");
    printf("Ctrl-C,and Ctrl-Break won’t generate a key ");
    printf("stroke.\nPress any key, %s to exit.\n", EXIT_KEY_NAME);
    ch = getch();
    if( ch == 0 )
       {
       printf("Cursor movenent or function key\n\n");
       getch();             /* remove 2nd part of key */
       }
    else
       printf("Key pressed is \"%c\"\n\n",(char)ch);
                            /* if flag has been set   */
    if( break_flag )
       {
       printf("\n\tCTRL-C/BREAK DETECTED\n");

/* NOTE: the application is responsible for resetting */
/*       the break flag after Ctrl-C/Break detected.  */
       break_flag = 0;
       }
    } while( ch != EXIT_KEY );
  printf("\n\nExit Key detected.  Exiting...\n");
  release();                /* uninstall the int 16h,
                               23h, and 1bh vectors   */
}

/*              -----  INFORM  -----
    tell the user what is going to happen
----------------------------------------------------- */
void inform()
{
int ch;

  printf("Ctrl-C and Ctrl-Break have been disabled.\n");
  printf("Press any key to see the keycode & character.\n");
  printf("Pressing Ctrl-C or Ctrl-Break will show that\n");
  printf("these keys have been trapped.\n\n");
  printf("Pressing Ctrl-2 or Alt-3 (keypad) will show\n");
  printf("^C and terminate the program.  Pressing %s\n", EXIT_KEY_NAME);
  printf("will exit the input loop and restore the\n");
  printf("interrupt vectors.\n\n");
  printf("Press any key to begin\n\n");
  ch = getch();             /* wait till key pressed  */
  if( ch == 0 )             /* if function or kursor  */
     ch = getch();          /* remove 2nd half        */
}

/* End of File */
```

Listing 11.3 `makefile`

```
##
##      makefile  -  Borland C makefile for test16.exe
##

EXEFILE = test16.exe
BC_LIB_DIR  = c:\borlandc\lib     # Borland C libraries
BC_INC_DIR  = c:\borlandc\include # Borland C includes
CR_INC_DIR  = .\                  # where errs.h is found

!ifdef DEBUG
DEBUG_SW = -v
!else
DEBUG_SW =
!endif

CC   = bcc -ms -c -w-par $(DEBUG_SW) -I$(CR_INC_DIR)
TASM = tasm  /MX /ZI /O
LINK = tlink $(DEBUG_SW) /L$(BC_LIB_DIR)

$(EXEFILE):   test16.obj ctrl_c.obj
        $(LINK) c0s test16 ctrl_c,\
                $(EXEFILE),nul, emu.lib maths.lib cs.lib;

test16.obj:   test16.c
        $(CC) test16.c

ctrl_c.obj:   ctrl_c.asm
        $(TASM) ctrl_c;

# End of File
```

Chapter 12

Handling DOS File Handle Limits

David L. Burki

Have you ever written a program that used more than three file handles at a time? Did you bother to check that the system would support more than three file handles before attempting to open the files or devices? Most people don't bother to check the number of available file handles unless the application needs a fairly large number, say eight or more. That's because the common approach to checking the number of available file handles is clumsy and slow. The common approach is to actually attempt to open a device (like NUL) as many times as the application needs file handles. If the all the opens are successful, your application can proceed. If they are not, you can suggest that the user increase the number of files in the `CONFIG.SYS FILES=` statement. In the case where the opens are successful, you have to close the device in order to return the handles to the system.

All these opens and closes take time. In this chapter, I'll show you a fast and easy way to:

- check the number of available file handles without opening a single file or device,
- increase the number of system-wide file handles available without modifying the `CONFIG.SYS` and rebooting,
- increase the maximum number of process file handles to 255 for DOS versions 3.1 and higher. The methods presented here do not rely on function `67h` of interrupt `21h`, which was introduced in DOS version 3.3. Relying on the functions present in DOS v3.3 and above could exclude many of your potential users.

When determining if a program (i.e., an individual process) can support the required number of open file handles, there are two limitations to consider:

- the number of file handles the system will support,
- the number of file handles each process can maintain at one time.

The number of system-wide file handles is set at boot time. The bootstrap process assures a minimum of eight. The normal number of handles a process can maintain is 20 because of the way DOS has been designed.

The user has control over the total number of file handles available to the system. While the basic design of DOS restricts the total number of file handles available for a process to 20. The user controls the number of system file handles through the `FILES=n` statement in the `CONFIG.SYS` file read at boot time. During the boot process, the initialization section of the bootstrap code searches the root directory of the boot volume for a

OFFSET (hex)	USE	SIZE
0	Address of next System File Handle Table node	Double Word
4	Number of individual File Information Blocks contained in this node	Word

Figure 12.1 ***System File Handle Table header layout.***

file named `CONFIG.SYS`. If a `CONFIG.SYS` file is found, the initialization code processes each line of the file and modifies the system based on the line. One of the possible directives in the `CONFIG.SYS` file is a `FILES=n` statement. If this line is present, the bootstrap code allocates and initializes one or more nodes in a linked list of System File Handle Table data structures. Each node in this linked list contains a header record (Figure 12.1) and one or more file handle information blocks (Figure 12.2). The header record contains a double word pointer to the next System File Handle Table node and an integer (2 bytes) which is the number of file handle information blocks in the node. DOS uses one file handle information block to manage handle access to each particular file/device. Without a `FILES=n` statement in the `CONFIG.SYS` or in the absence of a `CONFIG.SYS` file, the initialization section allocates only enough System File Handle Table nodes to accommodate eight files. Listing 12.8 shows how to expand the System File Handle Table from within your application. Using the techniques shown here means you'll never have to ask the user to edit the `CONFIG.SYS` file and reboot.

OFFSET (hex)	USE	SIZE (decimal)
0	Reference count	Word
2	Open mode	Word
4	Directory attributes	Byte
5	Flags	Word
7	Device control block (file)	Double Word
0B	Initial cluster (file)	Word
0D	Hours, Minutes, Seconds/2	Word
0F	Year, Month, Day	Word
11	File Size	Double Word
15	Current file pointer (position)	Double Word
19	Number of clusters from beginning of file	Word
1B	Current cluster	Word
1D	Block number	Word
1F	Directory index	Byte
20	File name	Byte[11]
2B	Unknown	Byte[4]
2F	Machine ID of file owner	Word
31	Segment of owner PSP	Word
32	Status	Word

Figure 12.2 ***System File Handle information block layout DOS v3.3.***

The following discussion and the code which accompanies this chapter make use of an undocumented DOS function, two undocumented internal DOS data structures, and three undocumented elements of the Program Segment Prefix (PSP). Table 12.1 lists the availability of these undocumented features. Because two elements which are required to expand the number of files for a process are not present in DOS v2.1, the following discussion of process file handles applies only to DOS v3.1 and higher. Remember the standard caveat about undocumented things in DOS: There is no guarantee that these things will remain constant in future releases of DOS.

The number of handles available to the system can be determined by examining the System File Handle Table linked list and adding up the number of handles each node can manage. Function `52h` of interrupt `21h` is an undocumented function which returns a pointer to the undocumented DOS List of Lists (Figure 12.3). The address of the root node of the System File Handle Table linked list is the second element (offset `4`) within the List of Lists data structure. The other entries in the List of Lists are defined only to stimulate your curiosity. The last System File Handle Table header record will contain `0FFFFh` as the offset portion of the pointer to the subsequent node, indicating the end of the list. The System File Handle Table linked list can be used to determine two important pieces of information: the total number of system-wide file handles

	DOS v2.1	**DOS v3.1 and higher**
Function 52h of interrupt 21h	yes	yes
System File Handle Table Header Record	yes	yes
System File Handle Table file information block	yes	yes
Process File Handle Table	yes	yes
Process File Handle Table Count	no	yes
Process File Handle Table Address	no	yes

Table 12.1 *Function and data availability.*

available and how many of those handles are already in use. Listing 12.1 contains the two callable functions which will provide this information. The function `max_sys_file_handles()` returns the total number of system-wide file handles available, and the function `sys_file_handles_in_use()` returns the number of system-wide file handles currently in use. `max_sys_file_handles()` simply walks the System File Handle Table linked list, accumulating the number of entries available in each node. The function `sys_file_handles_in_use()`, walks the same linked list but examines each file handle information block contained in the node and counts the number of information blocks actually in use. Listing 12.2 is a C language driver used to test this routine. Execute the driver with various values for `n` in the `CONFIG.SYS FILES=n` statement. The number of file handles available to the system will always equal the number of files specified in the `CONFIG.SYS`, the default of eight if no `FILES=` statement was processed from the `CONFIG.SYS`, or the default of eight if no `CONFIG.SYS` was found during boot. The value returned by `sys_file_handles_in_use()` will be three when executed from the command line. The three handles in use are the three device handles DOS passes to all processes when they are loaded. If you modify Listing 12.2 so that a few files or devices are

OFFSET (hex)	USE	SIZE
0	Address of Device Control Block linked list root node	Double word
4	Address System File Handle Table linked list root node	Double word
8	Address of the $CLOCK device driver	Double word
0C	Address of the CON device driver	Double word
10	Maximum sectior size	Word
12	Address of cache buffer linked list root node	Double word
16	Address of current directory information table	Double word
1A	Address of system FCB linked list root node	Double word
1E	Unknown	Word
20	Maximum number of drives	Byte
21	Current number of drives	Byte

Figure 12.3 ***DOS List Of Lists layout.***

opened prior to calling *sys_file_handles_in_use()*, the value returned will be the number of successful file opens plus the original three.

As I mentioned earlier, the number of file handles available to a process is limited by two things: the total number of unused file handles available in the system and the number of unused entries available in the file handle table used by the process. The difference between the values returned by the functions *max_sys_file_handles()* and *sys_file_handles_in_use()* (Listing 12.1) will tell how many handles are available at the system level. Once you have determined that there are enough unused handles available at the system level, the next step is to find out how many slots in the file handle table used by the process are already in use.

OFFSET (hex)	USE	SIZE (decimal)
0	Int 20h instruction	Word
2	Top of memory	Word
4	Reserved for DOS	Byte
5	Far call to DOS	Double Word
6	Number of bytes in segment (Also serves as the offset portion of the far call to DOS)	Word
0A	Int 20h terminate address	Double Word
0E	Address of Ctrl-Break handler to restore on exit	Double Word
12	Address of Critical Error handler to restore on exit	Double Word
* 16	Segment address of parent process PSP	Word
* 18	Default Process File Handle Table array	Byte[20]
2C	Segment address of the copy of the environment for this process	Word
* 2E	SS:SP save area used by int 21h handler to save invokers SS:SP	Double Word
* 32	Maximum number of file handles for this process	Word
* 34	Address of the Process File Handle Table	Double Word
38	Reserved for DOS	Byte[24]
* 50	DOS call (int 21h, retf)	Byte[2]
52	Reserved for DOS	Byte[10]
5C	FCB #1	Byte[16]
6C	FCB #2	Byte[20]
80	Command line parm length	Byte
81	Command line parameters	Byte[128]

Figure 12.4 *Program Segment Prefix layout.*

When DOS loads an executable program, part of the loader's job is to build a Program Segment Prefix (PSP). The PSP is a `100h`-byte data structure that precedes the program in memory. Many of the PSP elements are officially documented. Some of the more useful, undocumented elements have been described in books and magazines. Figure 12.4 shows the layout of the PSP. Fields marked with an asterisk are not officially documented. Three elements of the PSP are concerned with file handles for a process. The default Process File Handle Table is located at offset `18h` of the PSP. This is a 20-byte array in which DOS stores the index used to locate an individual file handle information block within the System File Handle Table linked list. The fixed size of this array is the reason each process is restricted to 20 file handles. A Process File Handle Table element is unused when it contains `0FFh`. The word located at offset `32h` within the PSP contains an integer which is the number of elements in the Process File Handle Table (normally 20). The third element of the PSP, and the one which will allow relocation of the Process File Handle Table, is a double word located at `34h` into the PSP. This is the address of the Process File Handle Table.

The function `process_handles_free()` (Listing 12.3) examines the Process File Handle Table, returning the number of entries that are unused. Listing 12.4 is a C language driver used to test `process_handles_free()`. You might modify the driver to open a few files prior to calling `process_handles_free()` to verify that the function does indeed return the number of free Process File Handle Table entries. When a program is loaded from the DOS command line, five entries in the Process File Handle Table are already taken. These entries are the handles for `stdin`, `stdout`, `stderr`, `stdprn`, and `stdaux`. These five handles are inherited from DOS when an application is started. In fact, these five handles are the ones DOS uses for its own I/O.

With the knowledge of the three file handle-related fields in the PSP and how the Process File Handle Table depends on the System File Handle Table list, there is nothing to prevent relocating and expanding the Process File Handle Table to allow more than 20 file handles. However, there is a warning when dealing with Process File Handle Table relocation. The DOS Load and Execute function (`4bh`, `INT 21h`) will not recognize a relocated file handle table. I suspect someone hard coded the address of the default Process File Handle Table when they wrote the Load and Execute function. If you plan to spawn a child process and want the child to inherit the parent's file handle table, you will have to make the child aware of the location and size of the parent's Process File Handle Table.

Listing 12.5 contains the assembly language functions necessary to relocate the Process File Handle Table. The equate which controls the number of elements in the relocated Process File Handle Table (`NEW_TABLE_SIZE`) has been set to 30. This equate should be changed to the number of file handles appropriate for your specific application. Each step of the relocation process has been made into a subroutine of its own for clarity. The function `relocate_pfht()` is the only externally callable function within Listing 12.5. The `relocate_pfht()` function simply calls the other routines in Listing 12.5 to relocate the Process File Handle Table. An enhanced version of `relocate_pfht()` might dynamically allocate the necessary space when passed an integer parameter which is the number of handles needed. Listing 12.6 is a C language driver used to validate `relocate_pfht()`.

The first thing `relocate_pfht()` does (after preserving a couple of registers using the handy `apush` macro) is call the function `find_pfht()`. The job of `find_pfht` is to return with the `ES:DI` register pair pointing to the new table, `CX` with the size of the new table, and the `DS:SI` register pair pointing to the current Process File Handle Table. Once addressability to the new table has been established, the segment and offset of the new table are loaded into the `ES` and `DI` registers. `CX` is loaded with the new table size established by the equate (`NEW_TABLE_SIZE`). With this accomplished, the next thing to do is locate the current Process File Handle Table. Locating the current Process File Handle Table is a two-part task. The first task is to locate the PSP. There are several methods which may be used to locate the PSP. Since the test driver is compiled under Microsoft C, I use the global variable, `_psp`. The Microsoft C compiler initializes this variable as the segment address of the PSP during the startup code. I suspect other high level language compilers provide a similar service. Check your documentation. If you are writing straight assembly code, the PSP is located at offset `0` from the segment address contained in the `DS` register when execution begins for `.exe` and `.com` files. Under DOS v3.x and higher, function `62h` of interrupt `21h` may also be used to establish the location of the PSP. Once the segment address of the PSP has been established in the `DS` register, the `lds si,[PFHT_ADDR]` instruction then loads the `DS:SI` register pair with the address of the current Process File Handle Table. Control is then returned to `relocate_pfht()`.

`relocate_pfht()` calls `init_new_pfht()` to initialize each entry in the new file handle table to `0FFh`, indicating an entry is unused. The new file handle table could be initialized by the assembler using `dup(0FFh)`, but explicitly initializing it in the code allows the buffer to be used for other things, if necessary, before you need to use it for the table. Initializing within the code also allows for alternate methods of storage allocation. `copy_old_to_new()` copies the current Process File Handle Table into the new, expanded table using the pointer to the Process File Handle Table found at offset `34h` of the PSP and the size of the table, found at offset `32h`. `update_pfht_count()` overwrites the old Process File Handle Table Count with the number of entries in the new table (`NEW_TABLE_SIZE`). `update_pfht_addr()` then changes the pointer to the Process File Handle Table (offset `34h`) to point to the newly established table. Listing 12.6 is a C program used to test the relocation process. The driver attempts to open the NUL device 30 times after relocating the Process File Handle Table. If the `FILES=n` statement in the `CONFIG.SYS` has been set to 30 or greater, this driver will fail when attempting to open the twenty-sixth handle. Remember, DOS passes the five process file handles when the process begins execution. Note that the call to create is a DOS interrupt `21h` function call. This is because Microsoft C has built in a limit of 20 file handles. Without modifying the startup code, MSC library functions (both stream and handle) will always fail when attempting to open the twenty-first file. A description of the modifications necessary to increase this limit can be found in the `readme` file accompanying the compiler.

The program shown in Listing 12.7 will check the number of available file handles. The timer routines used are included on the code disk as `tctimer.c` and `tctimer.h`. The timing test program illustrates the usage of the routines in Listings 12.1 and 12.2, as well as a typical alternative solution (opening the NUL device), to determine if 10 file handles are available. Execution times presented here are an average of the results obtained on a 10 MHz AT clone. The check took 0.74 milliseconds using the methods and code described in this chapter. The alternative solution (opening and closing the NUL device) required 21.31 milliseconds. By the way, the routines in Listing 12.1 are very inefficient. Execution speed could be increased if the number of system handles in use were calculated during the linked list traversal, which accumulates the total number of system-level file handles. They are separate functions here for clarity.

Every program which uses more than three file handles at a time should check the number of system-level file handles available, as well as the number of Process File Handle Table entries available as part of the program initialization. If there are not enough system-level handles available, the user can be informed that the `CONFIG.SYS FILES=` parameter needs to be increased or your program can expand the system-level handles as shown in Lising 12.8. If there are not enough process-level handles, you can decide if you want to relocate the Process File Handle Table or not.

Not many of us have need for 255 file handles. In fact, most of the time the 15 files available from DOS are enough. But when you need more than that, you *really* need the extra files. One of those exceptional occasions happened to me when I was modifying a disk-based merge sort and the source file to be sorted grew large enough to make it necessary to open a sixteenth work file. At the time I hadn't discovered the techniques presented here. I had to scrap the utility and resort to a commercial sort program. But sometimes a commercial solution is not available or isn't feasible. With the Process File Handle Table relocation code presented here, you can write the program yourself without worrying about DOS file handle limits.

Listing 12.1 sfht.asm

```
COMMENT #
------------------------------------------------------
  Non model specific method to determine the number
  of file handles available to the entire system.

  Written for Microsoft MASM 5.x to interface with
  Microsoft C 5.x.

  Module Contents:
        int max_sys_file_handles(void);
        int sys_handles_in_use(void);

  Author:
        David Burki     9/4/90  -  original version
------------------------------------------------------
END OF COMMENT #

.model small            ;memory model directive

;           --- EQUATES ---
FOXES           equ     0ffffh

;           --- STRUCTURES ---

; DOS List of Lists structure definition
list_of_list struc
   dcb_head_addr       dd  0  ;head DCB list
   sys_file_tbl_addr   dd  0  ;head SFHT list
   clock_addr          dd  0  ;clock device
   console_addr        dd  0  ;console device
   max_sector          dw  0  ;largest sector size
   cache_addr          dd  0  ;head of cache control block
   current_dir_addr    dd  0  ;current directory structure
   sys_fcb_tbl_addr    dd  0  ;head of sys file fcb list
   notknown            dw  0  ;unknown   borland tasm
   num_drives          db  0  ;specified by lastdrive=
   current_drive_cnt   db  0  ;current number of drives
list_of_list ends

; System File Handle Table Header structure definition
sys_file_tbl_hdr struc
   next_sys_file_tbl  dd    0  ;addr of next tbl in chain
   num_hdl_ctl_blks   dw    0  ;number of file handle control blks in this data structure
sys_file_tbl_hdr ends
```

Listing 12.1 *continued*

```
; System File Handle Information Block structure definition
sys_fcb struc
   open_cnt           dw   0  ;zero if not in use
   open_mode          dw   0  ;mode handle was open in
   dir_attr           db   0  ;directory attributes
   flags              dw   0  ;flags
   dbc                dd   0  ;device control block
   cluster1           dw   0  ;initial cluster
   time               dw   0  ;file time
   date               dw   0  ;file date
   fsize              dd   0  ;file size
   fpos               dd   0  ;file pointer location
   rel_cluster        dw   0  ;relative cluster
   cur_cluster        dw   0  ;current cluster
   blk_num            dw   0  ;block number
   dir_index          db   0  ;directory index
   file_name          db 11 dup(0) ;file/device name
   un_known           db  4 dup(0) ;unknown
   owner_machine      dw   0  ;machine id of owner
   owner_psp          dw   0  ;psp of owner process
   status             dw   0  ;status
sys_fcb ends

SYS_FILE_TBL_HDR_SIZE   equ     size sys_file_tbl_hdr
SYS_FCB_SIZE            equ     size sys_fcb

;                    --- CODE ---
.code

;-------------------------------------------------------
; int max_sys_file_handles(void);
;
;       Sums the number of individual file handle table
;       entries available to the system.
; CAUTION:
;   1.  Utilizes an undocumented function (52h) of
;       int 21h to obtain the addr of the List of Lists.
;   2.  Utilizes undocumented structures of the List of Lists,
;       the System File Handle Table Header Record and the
;       System File Handle Information Block data structures.
;   The availablity of function 52h and the layout of the
;   data structures is consistent for all DOS versions 2.1
;   and greater - at least as far as they are used here.

; INPUT:    None
; OUTPUT:   None
; RETURNS:  Number of file handles available to the
;           system in the AX register.
;           All other registers preserved.
;-------------------------------------------------------
```

Listing 12.1 *continued*

```
public _max_sys_file_handles
_max_sys_file_handles proc
; --- setup
        push    es
        push    bx
        push    ds
        push    si

; --- undocumented function to retrieve the
;     address of the list of lists (in es:bx)
        mov     ah,52h
        int     21h

; --- initialize (zero out) the counter
        xor     ax,ax

; --- establish addressability to the first
;     system file handle table data structure
        lds     si,es:[bx.sys_file_tbl_addr]

get_next:
; --- add the number of individual file handles
;     the data structure can accommodate
        add     ax,[si.num_hdl_ctl_blks]

; --- load the address of the next system file
;     handle table data structure in the chain
        lds     si,[si.next_sys_file_tbl]

; --- is the offset to the next 0ffffh, indicating that
;     the one just processed was the last in the chain
        cmp     si,FOXES

; --- if the last data structure has been processed
;     return to the caller
        jz      got_last

; --- else process the new system file handle table
;     data structure
        jmp     get_next

; --- clean up and return
got_last:
        pop     si
        pop     ds
        pop     bx
        pop     es
        ret
_max_sys_file_handles endp
```

Listing 12.1 *continued*

```
;-----------------------------------------------------
; int sys_handles_in_use(void);
;
;       Sums the number of DOS internal file handle
;       information blocks within the System File Handle
;       Table linked list that are in use.
; CAUTION:
;   1.  Utilizes an undocumented function (52h) of
;       int 21h to obtain the addr of the List of Lists.
;   2.  Utilizes undocumented structures of the List of Lists,
;       the System File Handle Table Header Record and the
;       System File Handle Information Block data structures.
;   The availablity of function 52h and the layout of the
;   data structures is consistent for all DOS versions 2.1
;   and greater - at least as far as they are used here.

; INPUT:    None
; OUTPUT:   None
; RETURNS:  Number of file handles currently in use in
;           the AX register.
;           All other registers preserved.
;-----------------------------------------------------

public _sys_handles_in_use
_sys_handles_in_use proc
; --- setup
        push    es
        push    bx
        push    ds
        push    si

; --- undocumented function to retrieve the
;     address of the list of lists
        mov     ah,52h
        int     21h

; --- initialize (zero out) the counter
        xor     ax,ax

; --- establish addressability to the first
;     system file handle table data structure
        lds     si,es:[bx.sys_file_tbl_addr]

get_next1:
; --- save the offset addr of the header record
        push    si

; --- load thge number of elements in this node
        mov     cx,[si.num_hdl_ctl_blks]

; --- bump the offset past the header record
        add     si,SYS_FILE_TBL_HDR_SIZE
```

Listing 12.1 *continued*

```
; --- is this handle control block in use?
nxt_element:
        mov     bx,[si.open_cnt]
        or      bx,bx
        jz      not_in_use
        inc     ax
not_in_use:

; --- move the offset on to the next element
        add     si,SYS_FCB_SIZE
        loop    nxt_element
        pop     si

; --- load the address of the next system file
;     handle table data structure in the chain
        lds     si,[si.next_sys_file_tbl]

; --- is the offset to the next 0ffffh, indicating that
;     the one just processed was the last in the chain
        cmp     si,FOXES

; --- if the last data structure has been processed
;     return to the caller
        jz      got_last1

; --- else process the new system file handle table
;     data structure
        jmp     get_next1

; --- clean up and return
got_last1:
        pop     si
        pop     ds
        pop     bx
        pop     es
        ret
_sys_handles_in_use endp

end

; End of File
```

Listing 12.2 testsfht.c

```
/* --------------------------------------------------


       TESTSFHT.C  -  Microsoft C ver 5.1

       A driver used to validate the assembly language
       functions max_sys_file_handles() and
       sys_handles_in_use().

       From the command line, executing this program
       will print the following line.
           Total system handles available: [n]
           System handles in use: [i]
       where n is the number of file handles specified
       in the CONFIG.SYS FILES=n directive and i is the
       number of system file information blocks which are
       active.  If no CONFIG.SYS file was found at boot
       time, or there was no FILES=n statement, or the
       FILES=n statement had n < 8, n will equal 8.  If
       run from the DOS command line without redirection,
       i will equal 3.  Output redirected to a file will
       increase i by 1.

       Test the driver with varying values n in the
       CONFIG.SYS FILES=n statement.  Remember to reboot
       after changing the CONFIG.SYS.

       Commands to produce an executable from the code as
       shown in the listings:
           masm sfht;             (listing 1)
           cl -c testsfht.c       (listing 2)
           link testsfht sfht;

  Author:
       David Burki     9/23/90  -  original version
   -------------------------------------------------- */

#include <stdio.h>

int max_sys_file_handles(void);
int sys_handles_in_use(void);

main()
{
  printf("Total system handles available: [%d]\n",
          max_sys_file_handles());
  printf("System handles in use: [%d]\n", sys_handles_in_use() );
  return 1;
}

/* End of File */
```

Listing 12.3 `pfht.asm`

```
COMMENT #
-------------------------------------------------------
  Determine the number of Process File Handle Table entries
  which are not currently in use.

  Written for Microsoft MASM 5.x to interface with
  Microsoft C 5.x.

  Module Contents:
        int process_handles_free(void);

  NOTE:
    This function makes use of two PSP fields (PFHT_ADDR and PFHT_COUNT)
    which are present only in DOS versions 3.0 and higher.

  Author:
        David Burki     9/20/90  -  original version
-------------------------------------------------------
END OF COMMENT #

.model small            ;memory model directive

;                  --- EQUATES ---

; --- an unused PFHT entry will contain 0ffh
FOXES           equ     0ffh

; --- offset within the PSP of the dword pointer to the PFHT
PFHT_ADDR       equ     34h

; --- offset within the PSP of the word containing the
;     number of elements in the PFHT
PFHT_COUNT      equ     32h

;                  --- DATA ---
.data

; --- declare MSC global variable containing the PSP segment address
extrn           __psp:word

;                  --- CODE ---
.code
```

Listing 12.3 *continued*

```
;------------------------------------------------------
; int process_handles_free(void);
;       Sums the number of elements within the Process File
;       Handle Table which are unused (contain 0ffh).
;
; INPUT:    None
; OUTPUT:   None
; RETURNS:  Number of file handles not in use in the
;           Process File Handle Table in the AX register.
;           All other registers preserved.
;------------------------------------------------------
public _process_handles_free
_process_handles_free proc
        push    ds
        push    si
        push    bx
        push    cx

; --- zero out the unused counter
        xor     bx,bx

; -- obtain the segment address of the psp
        mov     ax,DGROUP
        mov     ds,ax
assume ds:dgroup
        mov     ax,[__psp]
        mov     ds,ax
assume ds:nothing

; --- address the PFHT count and load cx with its value
        mov     si,PFHT_COUNT
        mov     cx,word ptr [si]

; --- address the pointer to the PFHT and load ds:si with
;     the dword pointer found
        mov     si,PFHT_ADDR
        lds     si,[si]

; --- setup ah for comparison to an unused entry in the PFHT
        mov     ah,FOXES
```

Listing 12.3 *continued*

```
; --- examine each element in the PFHT.  if it contains
;     0ffh it is unused so increment the unused count.
pfht_loop:
        lodsb               ;get an elements content
        xor     al,ah       ;is it unused
        jnz     not_empty   ;no, take the jump
        inc     bx          ;yes, inc the count
not_empty:
        loop    pfht_loop   ;go back for another

; --- move the total into ax for return to caller
        mov     ax,bx

        pop     cx
        pop     bx
        pop     si
        pop     ds
        ret
_process_handles_free endp

end

; End of File
```

Listing 12.4 `testpfht.c`

```
/* -------------------------------------------------

        TESTPFHT.C  -  Microsoft C ver 5.1

        A driver used to validate the assembly language
        function process_handles_free().

        From the command line, executing this program
        will print the following line.
             free handles: [15]

        When executed under Microsoft CodeView ver 4.0,
        this program will print the following line
             free handles: [14]
        since CodeView has taken one PFHT entries for
        its own use (Turbo Debugger doesn't take any).

        Further proof of the function can be obtained by
        opening one or more files, calling the function
        to check the free handles, and then closing one
        file at a time. After each close, call the function.
        The free handle count will increase with each close.

        Commands to produce an executable from the code as
        shown in the listings:
            masm pfht;              (listing 3)
            cl -c testpfht.c        (listing 4)
            link testpfht pfht;

   Author:
        David Burki     9/23/90  -  original version
        David Burki    12/11/93  -  update to MSC 7 & BC 3.1
   ------------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>      /* for _psp definition */

int process_handles_free( void );

main()
{
   printf("free handles: [%d]\n", process_handles_free());
   return 0;
}

/* End of File */
```

Listing 12.5 `relocate.asm`

```
COMMENT #
------------------------------------------------------
  Relocate the Process File Handle Table to the area defined
  within this module (new_table). This will allow the number
  of handles for this process to be expanded up to 255. The
  equate NEW_TABLE_SIZE defined below should be adjusted
  to reflect the application specific handle requirements.

  Written for Microsoft MASM 5.x to interface with
  Microsoft C 5.x.

  Module Contents:
        void relocate_pfht(void);       - public
        void find_pfht(void);           - local
        void init_new_pfht(void);       - local
        void copy_old_to_new(void);     - local
        void update_pfht_count(void);   - local
        void update_pfht_addr(void);    - local

  NOTE:
    The function find_pfht() makes use of two PSP fields
    (PFHT_ADDR and PFHT_COUNT) which are present only in
    DOS versions 3.0 and higher.

  Author:
        David Burki     09/22/90  -  original version
------------------------------------------------------
END OF COMMENT #

.model small            ;memory model directive

;                  --- MACROS ---

;push selected registers
apush   macro   a,b,c,d,e,f,g,h
        irp x,<a,b,c,d,e,f,g,h>
        ifnb <x>
        push    x
        endif
        endm
        endm

;pop selected registers
apop    macro   a,b,c,d,e,f,g,h
        irp x,<h,g,f,e,d,c,b,a>
        ifnb <x>
        pop     x
        endif
        endm
        endm
```

Listing 12.5 *continued*

```
;                  --- EQUATES ---

; --- this equate should be adjusted to the number of
;     file/device handles required by your application.
NEW_TABLE_SIZE          equ     30

; --- offset within the PSP of the dword pointer to the PFHT
PFHT_ADDR               equ     34h

; --- offset within the PSP of the word containing the
;     number of elements in the PFHT
PFHT_COUNT              equ     32h

; --- an unused PFHT entry will contain 0ffh
FOXES                   equ     0ffh

;                  --- DATA ---
.data

; --- this is the data area for the expanded PFHT
public _new_table
_new_table       db       NEW_TABLE_SIZE dup(0);

; --- declare MSC global variable containing the PSP segment address
extrn           __psp:word

;                  --- CODE ---
.code

;----------------------------------------------
; void relocate_pfht(void);
;       Move the current PFHT to the area allocated in the
;       data segment above.
; TRASHES:
;       AX
;----------------------------------------------
public _relocate_pfht
_relocate_pfht proc
        apush   ds,si,es,di,cx
        call    find_pfht
        call    init_new_pfht
        call    copy_old_to_new
        call    update_pfht_count
        call    update_pfht_addr
        apop    ds,si,es,di,cx
        ret
_relocate_pfht endp
```

Listing 12.5 *continued*

```
;-----------------------------------------------
; local void find_pfht(void)
;       setup registers with the current PFHT, the
;       new PFHT and the size of the current PFHT.
; NOTE:
;       Uses the global variable __psp defined and
;       initialized by MSC startup code.
; RETURNS:
;       DS:SI pointing to the current PFHT
;       ES:DI pointing to the new, expanded PFHT
;       CX   size of the current PFHT
; TRASHES:
;       AX
;-----------------------------------------------
find_pfht proc
; --- address the MSC data segment
        mov     ax,DGROUP
        mov     ds,ax
assume ds:DGROUP

; --- get the offset of the new table
        lea     di,_new_table

; --- get the segment of the new table into es
        mov     ax,seg _new_table
        mov     es,ax

; --- get the segment of the PSP in ds
        mov     ds,[__psp]
assume ds:nothing

; --- load cx with the current PFHT count
        mov     cx,word ptr ds:[PFHT_COUNT]

; --- address of current PFHT into ds:si
        lds     si,ds:[PFHT_ADDR]
        ret
find_pfht endp
```

Listing 12.5 *continued*

```
;-----------------------------------------------
; local void init_new_pfht(void)
;       initialize the new PFHT to 0FFh.
; PRESERVES:
;       DI and CX
; TRASHES:
;       AX
;-----------------------------------------------
init_new_pfht proc
        apush   di,cx

; --- get the new table size
        mov     cx,NEW_TABLE_SIZE

; --- load al with 0FFh
        mov     al,FOXES

; --- move 0FFh into every byte of the table
        repnz   stosb

        apop    di,cx
        ret
init_new_pfht endp

;-----------------------------------------------
; local void copy_old_to_new(void)
;       Copies the contents of the current PFHT to the new PFHT.
; ON ENTRY:  DS:SI points to current PFHT
;            ES:DI points to new, expanded PFHT
;            CX    contains the number of entries (bytes)
;                  in the current PFHT
; PRESERVES: DI
; TRASHES:   CX
;-----------------------------------------------
copy_old_to_new proc
; --- save the offset of the new table
        push    di
        repnz   movsb
        pop     di
        ret
copy_old_to_new endp
```

Listing 12.5 *continued*

```
;------------------------------------------------
; local void update_pfht_count(void)
;       Update the PFHT Count contained in the PSP to
;       reflect the size of the new table.
; ON ENTRY:  DS    segment of PSP
; TRASHES:
;            AX, SI
;------------------------------------------------
update_pfht_count proc
; --- load the size of the new table
        mov     ax,NEW_TABLE_SIZE

; --- point to the count word and store the new count
        mov     si,PFHT_COUNT
        mov     word ptr [si], ax
        ret
update_pfht_count endp

;------------------------------------------------
; local void update_pfht_addr(void)
;       Update the pointer to the PFHT contained in the PSP
;       to point to the new PFHT.
; ON ENTRY:  ES:DI points to the new, expanded PFHT
;            DS    segment of PSP
; TRASHES:
;            SI
;------------------------------------------------
update_pfht_addr proc
; --- point to the lo word of the address
        mov     si,PFHT_ADDR

; --- store the offset of the new table
        mov     [si],di

; --- store the segment of the table
        mov     [si+2],es
        ret
update_pfht_addr endp

end

; End of File
```

Listing 12.6 `testrelo.c`

```
/* ----------------------------------------------------
        TESTRELO.C  -  Microsoft C ver 5.1

        A driver used to validate the assembly language
        function relocate_pfht().

        NOTE:
           The routines open_file() and close_file() are
           included here because the limit imposed by
           Microsoft on the number of open streams and/or
           handles using their library functions.

        Commands to produce an executable from the code as
        shown in the listings:
            masm relocate;          (listing 5)
            cl -c testrelo.c        (listing 6)
            link testrelo relocate;

  Author:
        David Burki     9/23/90  -  original version
  ---------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>     /* errno definition */
#include <dos.h>        /* file create attributes */

/* define the maximum attempts to open NUL */
#define MAX_TRIES 30

               /* FUNCTION PROTOTYPES */
void useage(void);
int open_file(char far *, unsigned);
int close_file(unsigned);
extern int relocate_pfht(void);

/* a place to store the handles returned by open_file() */
int handles[MAX_TRIES];
```

Listing 12.6 *continued*

```
/* -----------------------------------------------------
 *  MAIN()
 ---------------------------------------------------------- */
int main( int argc, char *argv[] )
{
int i, num;
                    /* handle some simple exceptions */
  if( argc < 2 || *(argv[1]) == '?' )
     {
     useage();
     exit(1);
     }

  num = atoi(argv[1]);
  if( num > 30 )
     {
     useage();
     exit(1);
     }

  relocate_pfht();  /* move the PFHT */

                    /* attempt to open the NUL device */
  for(i = 0; i < num; i++)
     {
     handles[i] = open_file((char far *)"nul", _A_NORMAL);
     if( handles[i] == -1 )
        {
        printf("open not successful.  errno = %x hex\n", errno);
        printf("failed attempting to open #%d\n", i+1);
        break;
        }
     }
                    /* close what was opened */
  while( i >= 0 )
     close_file(handles[i--]);
  return 0;
}
```

Listing 12.6 *continued*

```
/* ----------------------------------------------------
 *  USEAGE()
 * Let the user know what is expected as input and what
 * the output will be.
   ----------------------------------------------------  */
void useage()
{
  printf("Pass a command line parameter less than or\n");
  printf("equal to %d.  This driver will then attempt\n", MAX_TRIES);
  printf("to open the NUL device %d times.\n\n", MAX_TRIES);
  printf("The driver will report the number on which\n");
  printf("open failed.\n");
}

/* ----------------------------------------------------
 *  OPEN_FILE()
 * use function 0x3c, int 0x21 to open a file/device
 * PARAMETERS:
 *      name    - pointer to the file/device name
 *      attr    - open mode attributes (same as MSC
 *                function open())
 * RETURNS:
 *      success:  the handle to the file/device
 *      failure:  -1 and global variable errno set to
 *                the error code returned by DOS.
 * NOTE the far pointer to the filespec.
   ----------------------------------------------------  */
int open_file( char far *name, unsigned attr )
{
union REGS inregs, outregs;
struct SREGS segs;
unsigned ret_val;
                    /* setup for DOS open call */
  segs.ds = FP_SEG(name);
  inregs.x.dx = FP_OFF(name);
  inregs.h.ah = 0x3c;
  inregs.x.cx = attr;
                    /* DOS open */
  int86x(0x21, &inregs, &outregs, &segs);

  if( outregs.x.cflag )
     {              /* open failed */
     errno = outregs.x.ax;
     ret_val = -1;
     }
  else              /* open successful */
     ret_val = outregs.x.ax;

  return(ret_val);
}
```

Listing 12.6 *continued*

```
/* ----------------------------------------------------
 *  CLOSE_FILE()
 * use function 0x3e, int 0x21 to close a file/device
 * PARAMETERS:
 *      handle  - the handle of the open file/device
 * RETURNS:
 *      success:  zero
 *      failure:  -1 and global variable errno set to
 *                the error code returned by DOS.
  ---------------------------------------------------*/
int close_file( unsigned handle )
{
union REGS inregs, outregs;
unsigned ret_val;
                  /* setup for the DOS close call */
  inregs.h.ah = 0x3e;
  inregs.x.bx = handle;
                  /* DOS close */
  int86(0x21, &inregs, &outregs);
                  /* failed close */
  if( outregs.x.cflag )
     {
     errno = outregs.x.ax;
     ret_val = -1;
     }
  else            /* successful close */
     ret_val = 0;

  return(ret_val);
}

/* End of File */
```

Listing 12.7 testtime.c

```
/* ----------------------------------------------------
        TESTTIME.C  -  Microsoft C ver 5.1

        A program to determine execution times. Displays
        the number of milliseconds needed to determine if
        enough file handles are available for an application
        to proceed using the functions in Listings 1 and 2
        as well as by opening the NUL device.

        The number of handles which must be available for
        "success" is determined by the #define below

        Commands to produce an executable from the code as
        shown in the listings:
            masm sfht;             (listing 1)
            masm pfht;             (listing 3)
            masm timer;            (not included)
            cl -c ctimer.c         (not included)
            cl -c testtime.c       (listing 7)
            link testtime sfht pfht timer ctimer;

  Author:
        David Burki     9/23/90  -  original version
   -------------------------------------------------- */
#include <stdio.h>
#include <dos.h>
#include "tctimer.h"
#include <io.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>

/*
 * how many file handles must be available for the
 * "application" to continue
*/
#define HANDLES_I_NEED 10

/*
; prototypes
*/
int max_sys_file_handles(void);
int sys_handles_in_use(void);
int process_handles_free(void);
```

Listing 12.7 *continued*

```
/* --------------------------------------------------
 * MAIN()
   --------------------------------------------------  */
main()
{
long t1,t2;
double time;
int sys_handles, sys_handles_used,
    sys_available, process_free,
    ok_to_proceed, h_table[HANDLES_I_NEED], i;

  initializetimer();    /* init the timer routines */

/* Listings 1 & 2 method */
  printf("\nusing routines from listings 1 & 2\n");

  t1 = readtimer();     /* get start time */

  ok_to_proceed = 1;    /* default = there are enough */

  sys_handles = max_sys_file_handles();
  sys_handles_used = sys_handles_in_use();
  sys_available = sys_handles - sys_handles_used;
  process_free = process_handles_free();
/*
 * if there are not enough system-level handles or not
 * enough process-level handles available, indicate
 * that the application can't proceed
*/
  if( process_free < HANDLES_I_NEED ||
      sys_available < HANDLES_I_NEED )
     ok_to_proceed = 0;

  t2 = readtimer();     /* get end time */
                        /* calculate elapsed time */
  elapsedtime(t1, t2, &time);
  printf("elapsed time = %f\n%s\n\n",
         time, (ok_to_proceed==1) ? "OK" : "NOT OK");

/* "open the nul device" method */
  printf("using open()\n");

  t1 = readtimer();     /* get start time */

  ok_to_proceed = 1;    /* default = there are enough */
```

Listing 12.7 *continued*

```
  for(i = 0; i < HANDLES_I_NEED; i++)
     {                 /* if the open failed */
     if( (h_table[i] = open("nul", O_TEXT | O_RDWR)) == -1 )
        {              /* indicate failure & leave loop */
        ok_to_proceed = 0;
        break;
        }
     }
  while( i >= 0 )      /* close what was opened */
     close(h_table[i--]);

  t2 = readtimer();     /* get end time */
                        /* calculate elapsed time */
  elapsedtime(t1, t2, &time);
  printf("elapsed time = %f\n%s\n\n",
         time, (ok_to_proceed==1) ? "OK" : "NOT OK");
                        /* clean up the timer routines */
  restoretimer();

  return 0;
}

/* End of File */
```

Listing 12.8 `inc_sfht.c`

```
/* ---------------------------------------------------
    inc_sfht.c

    Make additional entries available in the System File
    Handle Table. Use this in lieu of forcing the user to
    add a FILES= statement in the config.sys and re-boot.

    Note the use of large model. Data pointers are all
    far. Large model permits use of regular library
    string functions. For small or compact use the
    far counterparts of the runtime library functions.

    To build, see listing8.mak

    ------------------------------------------------- */

#include <stdio.h>
#include <alloc.h>
#include <dos.h>
#include <string.h>

// NOTE: structs must be packed on byte boundary
#pragma pack(1)

typedef struct list_of_lists
{
  void far      *dcb_head;
  void far      *handle_sft;
  void far      *clock;
  void far      *console;
  unsigned      max_sector;
  void far      *cache;
  void far      *cur_dir;
  void far      *fcb_sft;
  unsigned      unknown;
  unsigned char drive_cnt;
  unsigned char lastdrive;
} LOL, far *PLOL;

typedef struct system_file_table_header
{
  void far *next;
  unsigned count;
} SFTH, far *PSFTH;

#define SFTH_SIZE sizeof(SFTH)
```

Listing 12.8 *continued*

```
// prototypes
short open_files( void );
short add_system_handles( short );
short find_size( char far * );
void  remove_added_handles( void );
void  close_files( void );

// a place to hold the original end-of-list header
static PSFTH old_end_of_list = (PSFTH)0;

// for the testing file structures
FILE *fid_array[20];

/* ---------------------------------------------------
 *  MAIN()
   --------------------------------------------------- */
main()
{
  printf("opened %d before expansion\n", open_files());
  close_files();
  add_system_handles( 5 );
  printf("opened %d after expansion\n", open_files());
  close_files();
  remove_added_handles();
  printf("opened %d after reset\n", open_files());
  close_files();
  return 0;
}

/* ---------------------------------------------------
 *  OPEN_FILES()
   --------------------------------------------------- */
short open_files()
{
  short i;

  for ( i = 0; i < 20; i++ )
    if ( (fid_array[i] = fopen("nul", "wt")) == (FILE *)0 )
      break;
  return i;
}

/* ---------------------------------------------------
 *  CLOSE_FILES()
   --------------------------------------------------- */
void close_files()
{
  short i = 0;

  while ( fid_array[i] != (FILE *)0 )
    fclose(fid_array[i++]);
}
```

Listing 12.8 *continued*

```
/* ----------------------------------------------------
 *  ADD_SYSTEM_HANDLES()
   ----------------------------------------------------  */
short add_system_handles( short new_entries )
{
  union REGS inregs,
             outregs;
  struct SREGS sregs;
  PLOL  lol;
  PSFTH sfth,
       new_block;
  char far *sft;
  short infoblk_size;

  if ( old_end_of_list != (PSFTH)0 )
    return -1;

  // get the list of lists address
  inregs.h.ah = 0x52;
  int86x(0x21, &inregs, &outregs, &sregs);
  FP_SEG(lol) = sregs.es;
  FP_OFF(lol) = outregs.x.bx;

  // get the first table header address
  sfth = lol->handle_sft;

  // get the actual table address
  sft = (char far *)sfth + SFTH_SIZE;

  // find the size of a single file information block
  infoblk_size = find_size(sft);
  if( infoblk_size == 0 )
    return -2;

  // find the last node in the SFHT
  while ( FP_OFF(sfth->next) != 0xffff )
    sfth = sfth->next;

  // chain on some more entries & save the old end address
  new_block = (PSFTH)calloc(new_entries, infoblk_size + SFTH_SIZE);
  if ( new_block == (PSFTH)0 )
    return 0;

  old_end_of_list = sfth;

  // init the new entry header
  FP_OFF(new_block->next) = 0xffff;
  new_block->count = new_entries;
```

Listing 12.8 *continued*

```
  // chain it in and return success
  sfth->next = new_block;
  return 1;
}

/* -----------------------------------------------------
 *  REMOVE_ADDED_HANDLES()
   ----------------------------------------------------- */
void remove_added_handles()
{
  // if haven't allocated, don't free
  if ( old_end_of_list == (PSFTH)0 )
    return;

  // free allocated memory
  free(old_end_of_list->next);

  // make the original end of list header the end again
  FP_OFF(old_end_of_list->next) = 0xffff;

  // clear the static pointer
  old_end_of_list = (PSFTH)0;
}

/* -----------------------------------------------------
 *  FIND_SIZE()
   ----------------------------------------------------- */
short find_size( char far *ft_ptr )
{
  char  far *ch_ptr, far *aux_ptr;
  short i;

  ch_ptr = ft_ptr;

  // search for the string "AUX" (1st info blk entry)
  // limit the search to 100 bytes into the STHT
  for ( i = 0; i < 100; i++ )
  {
    if ( strncmp(ch_ptr, "AUX", 3) == 0 )
    {
      aux_ptr = ch_ptr;
      break;
    }
    else
      ch_ptr++;
  }
```

Listing 12.8 *continued*

```
  // if not found, return error
  if ( i == 100 )
    return 0;

  // now search for "CON" (2nd info blk entry)
  // limit the search to 100 bytes into the STHT
  for ( i = 0; i < 100; i++ )
  {
    if ( strncmp(ch_ptr, "CON", 3) == 0 )
      break;
    else
      ch_ptr++;
  }

  // if not found, return error
  if ( i == 100 )
    return 0;

  // return the difference (i.e. size of an info blk)
  return (short)(ch_ptr - aux_ptr);
}

/* End of File */
```

Chapter 13

Preventing Lost Keystrokes

David L. Burki

Preventing DOS From Losing Keystrokes

The PC keyboard interface has undergone significant changes since it was first introduced. Keyboards now have more keys and the BIOS supplies new functions for accessing them. Although the software interface to new keyboards is almost backwardly compatible, there are a few glitches. One of those glitches is that, under the right conditions, DOS may lose keystrokes. After a quick review of keyboard basics, this chapter describes how DOS loses keystrokes and provides a TSR (Terminate-and-Stay-Resident program) that corrects the problem.

Keyboard Basics

The BIOS maintains a 15-key internal buffer (BIOS keyboard buffer) where the BIOS keyboard hardware interrupt (`INT 9`) stores keystrokes received from the keyboard and from which programs (including DOS) may retrieve keystrokes utilizing the BIOS keyboard interrupt (`INT 16h`). The BIOS stores keystrokes as word values within the BIOS keyboard buffer. The high byte of each word (key code) contains a scan code that uniquely identifies a physical key on the keyboard and, if the keystroke was an ASCII character, the low byte contains the ASCII code.

When IBM introduced the original PC, the keyboard interface was fairly simple. A scan code generated by a key press or release was transferred to the keyboard controller, read by the BIOS keyboard hardware interrupt (`INT 9`), and placed into the BIOS keyboard buffer. The high byte of the key code generally contained the original scan code sent by the keyboard and, if the key was an ASCII character, the low byte contained the ASCII code corresponding to the key that was pressed. When DOS or an application wanted a character from the keyboard, a call to `INT 16h` returned the key code at the head of the BIOS keyboard buffer.

With the introduction of the AT, IBM gave us a keyboard with 84 keys (IBM added *SysReq*) and a different key layout. The AT keyboard produces scan codes different from the original PC keyboard because physical circuit design efficiency dictates scan code assignment by key location. For software compatibility, IBM provided a scan code translation within the keyboard controller. After this translation step, the scan code for a given key is identical to that key's scan code on a PC. Because this translation step happens before the BIOS gets a look at the scan code, the BIOS `INT 9` handler (and all higher level software) sees the same set of scan codes as they did with the original PC.

The latest keyboard from IBM is the 101-key enhanced keyboard. The most noticeable difference in the enhanced keyboard is the addition of a separate set of cursor control (duplicate editing) keys and the addition of two more function keys. This keyboard has yet another key layout, and instead of coming up with another set of scan codes, IBM brought out three sets. Scan code set 1 produces scan codes compatible with the original PC keyboard; scan code set 2 produces the AT and enhanced keyboard scan codes; and scan code set 3 is used for the IBM RT. The active scan code set can be selected by software and, at boot time, the BIOS makes scan code set 2 the active scan code set.

For the most part, scan code set 2 produces scan codes equivalent to the 84-key keyboard. The major difference between the AT and enhanced keyboards is that the enhanced keyboard produces scan codes for the new keys. Instead of giving additional translation capabilities to the keyboard controller, `INT 16h` handles these new scan codes. BIOS versions that support the enhanced keyboard have four added functions in `INT 16h`. Three of the added functions, `10h`, `11h`, and `12h` are the extended keyboard versions of the original `INT 16h` functions `0` ("get key"), `1` ("get key status"), and `2` ("get shift status"). The other added function in `INT 16h` is function `5`, which places the key code in the `CX` register into the BIOS keyboard buffer just as if it had been typed at the keyboard.

When a program (including DOS) calls `INT 16h` with one of the old function values `0` or `1`, and the key code at the head of the BIOS keyboard buffer represents one of the duplicate editing keys, the BIOS graciously translates the key code to the value it would have been had the corresponding editing key on the PC or AT keyboard been pressed. If the next available key code represents a new key (*F11* or *F12*) or a key combination not recognized by the original functions (`Ctrl-insert` or numeric keypad `5` for example), `INT 16h` throws away the key code by advancing the pointer toward the BIOS keyboard buffer.

Generally, the translation and removal of key codes is not a problem, except that DOS always uses the old function calls to examine or get a key. During certain functions of `INT 21h`, DOS checks the next available key in the keyboard buffer to see if *Ctrl-C* was pressed. (See Chapter 11 "*Ctrl-C* and *Ctrl-Break* Handling under MS-DOS.") By using function `1` ("get key" status) to check the keyboard buffer, DOS causes `INT 16h` to remove the key codes that the old `INT 16h` function calls would not recognize. The removal or translation becomes a problem when, for example, your program is correctly using the BIOS extended keyboard function calls and does a DOS `Write String` (`INT 21h`, function `9`) while one of the new key codes is at the head of the buffer. The key code will not be there when DOS returns to your program. If your application is using these key codes and uses DOS function calls, there is a high probability that at least some of these key codes will never make it to your program.

The code in Listing 13.1 is a TSR program to translate function requests that could destroy key codes in the BIOS keyboard buffer. The TSR converts `INT 16h` functions `0` and `1` into functions `11h` and `12h`. This prevents any program (including DOS) from causing the BIOS to throw away buffered key codes. Although every program should contain code to handle unknown keystrokes, many do not and may behave unexpectedly with this TSR installed. You will have to experiment to determine which programs will behave properly.

Listing 13.1 EXTND16.ASM

```
COMMENT ~
------------------------------------------------------------
This is a Terminate & Stay Resident program which when
installed will intercept INT 16h calls and insure that
function calls zero and one are changed to functions 10h
and 11h.

The TSR consumes about 380 bytes when installed.

To create an executable, assemble & link this module and
convert the .EXE to a .COM with exe2bin.

Author:  David Burki
------------------------------------------------------------
END OF COMMENT ~

;               ----------  EQUATES  ----------
ENV_OFFSET      equ     02ch
GET_KEY         equ     0
CHK_FOR_KEY     equ      1
X_STUFF_KEY     equ     5
X_GET_KEY       equ     10h
X_CHK_FOR_KEY   equ     11h

;                 ----------  CODE  ----------
cseg segment byte public 'code'

; --- com file setup
        org     100h

assume cs:cseg, ds:nothing, es:nothing

entry:
        jmp     initialize

; --- storage for the int 16h vector that's intercepted
orig_16_vect    dd      ?
```

Listing 13.1 *continued*

```
;--------------------------------------------------------
; PRE_16()
;   This function gets to look at the INT 16h function call
;   and change function calls 0 & 1 to 10h & 11h so that
;   DOS can't destroy keycodes when it checks the status of
;   the BIOS kbd buffer.
;
; Caution:  Some programs (CED for example) will not
; correctly interpert the keycodes returned from the
; extended keypad keys.  Strange characters appear when
; these keys are pressed.
;--------------------------------------------------------
pre_16 proc far
; --- save the flags we received & interrupts off
        pushf
        sti

; --- if function is greater than 1 (status req) don't do
;     anything - just jump to the original vector
        cmp     ah,1
        ja      go_orig_16

; --- otherwise, make function 0 into 10h & 1 into 11h
        or      ah,10h

; --- restore callers flags & pass on to original vector
go_orig_16:
        popf
        jmp     orig_16_vect
pre_16 endp

; --- anything after this label is init code & will be
;     thrown away after installing
end_resident label byte

assume cs:cseg, ds:cseg, es:cseg
```

Listing 13.1 *continued*

```
;----------------------------------------------------------
; FREE_ENVIRONMENT()
;   Release this program's copy of the DOS environment in
;   order to make this TSR as small as possible.
;
; Freeing the environment makes it impossible for utilities
; like MAPMEM to determine what program owns the memory
; allocated to this resident process.
;
; Destroys:
;       AX
;----------------------------------------------------------
free_environment proc near
        push    es
        push    si
        xor     ax,ax

; --- get environment segment from the PSP
        mov     si,ENV_OFFSET

; --- move segment addr of environment into AX & at same
;     time move the zero that's in AX into the PSP pointer
;     to the environment
        xchg    ax,[si]
        or      ax,ax
        jz      environment_exit
        mov     es,ax
        mov     ah,49h
        int     21h
environment_exit:
        pop     si
        pop     es
        ret
free_environment endp
```

Listing 13.1 *continued*

```
;---------------------------------------------------------
; CK_BIOS_SUPPORT
;   Check the BIOS to see if enhanced functions are
;   available by attempting to use one of the enhanced
;   functions - function 5, add keycode to BIOS keyboard
;   buffer.
; Returns:
;   Carry clear - BIOS supports enhanced functions
;   Carry set   - BIOS does not support enhanced
;                 functions
;---------------------------------------------------------
ck_bios_support proc near
; --- flush the BIOS kbd buffer
        mov     ah,CHK_FOR_KEY
        int     16h
        jz      buf_is_empty
        mov     ah,GET_KEY
        int     16h
        jmp     ck_bios_support

; --- once the buffer is empty, attempt to add a keycode
;     to the buffer
buf_is_empty:
        mov     ah,X_STUFF_KEY
        mov     cx,0ffffh
        int     16h

; --- if AL comes back non-zero add keycode failed
        or      al,al
        jnz     no_support

; --- just to make sure, see if the keycode at the head
;     of the BIOS kbd buffer is the one we added
        mov     ah,X_CHK_FOR_KEY
        int     16h
```

Listing 13.1 *continued*

```
; --- is the buffer empty?
        jz      no_support

; --- no, remove the keycode we stuffed with an extended
;       get-key function call
        mov     ah,X_GET_KEY
        int     16h

; --- is it the same keycode
        cmp     ax,0ffffh
        je      yes_support

; --- exit indicating no support
no_support:
        stc
        jmp     ck_bios_exit

; --- exit indicating support
yes_support:
        clc

ck_bios_exit:
        ret
ck_bios_support endp

no_install_msg  db      'BIOS does not support extended '
                db      'keyboard functions.',0dh,0ah,'$'

install_msg     db      'Installing extended BIOS keyboard'
                db      ' function support.',0dh,0ah,'$'
```

Listing 13.1 *continued*

```
;---------------------------------------------------------
; INITIALIZE()
;   Check that enhanced BIOS keyboard support is
;   available and install only if it is.
;---------------------------------------------------------
initialize proc near
        call    free_environment
        call    ck_bios_support

; --- if extended BIOS kbd support not available don't
;     install
        jc      no_install

; --- display the install message
        mov     dx,offset install_msg
        mov     ah,9
        int     21h

; --- get the current INT 16h vector & sace it in a code
;     segment variable
        mov     ax,3516h
        int     21h
        mov     word ptr orig_16_vect,bx
        mov     word ptr orig_16_vect+2,es

; --- put pre_16() in the vector table
        mov     dx,offset pre_16
        mov     ax,2516h
        int     21h

; --- calculate the length of the resident portion in
;     bytes and round up to next paragraph
        mov     dx,offset end_resident
        add     dx,15

; --- convert bytes to paragraphs
        mov     cl,4
        shr     dx,cl

; --- terminate & stay resident, DX has the size of
;     resident portion in paragraphs
        mov     ah,31h
        int     21h
```

Listing 13.1 *continued*

```
; --- display the non-installation message & terminate
;     the process without staying resident
no_install:
        mov     dx,offset no_install_msg
        mov     ah,9
        int     21h
        mov     ax,4c01h
        int     21h
initialize endp

cseg ends

end entry

; End of File
```

Chapter 14

Gracefully Breaking Out of Infinite Loops

David L. Burki

At some point in almost every programmer's career, he or she codes a looping construct that will never terminate. Those of us who write in assembly or C have the added excitement of being able to write over code, or to move off and execute in some random location in memory. Getting out of such situations generally requires rebooting the machine, especially when the loop contains no DOS (`Int 21h`) function calls that check the status of the DOS *Ctrl-C/Break* flag. In this article, I will show how to provide for a graceful exit when your program hangs on an infinite loop or is executing garbage.

While searching for ways to dispose of the `^C` displayed by *Ctrl-C* and *Ctrl-Break* (see Chapter 11), I came across a bit flag in the BIOS data area that is set when `Int 9` (the BIOS keyboard hardware interrupt) detects the *Ctrl-Break* key combination. Once I had all the pieces to the *Ctrl-C/Break* puzzle in place, it was apparent that the BIOS break flag didn't play a part. However, the flag stimulated some thought and resulted in the routines described here. These routines are intended as tools for use during development, not as part of a marketable product. The tools' objectives are simple: provide a set of routines that will detect the *Ctrl-Break* key combination, call a clean-up routine if necessary, and terminate the "hung" application.

Determining Machine Stability

You can use the breakout tool to interrupt most any process, simply by pressing the designated hot key. The process might be your program, DOS, or a BIOS interrupt. Since the hot key can interrupt almost any task, the breakout tool must know that the machine state is stable before it successfully terminates the application. For the purposes of the breakout tool, the following conditions must exist for the machine state to be considered stable:

- No disk I/O in progress.
- DOS must be in a state in which DOS functions (*Int 21h*) can be made.
- No critical error in progress.

To determine whether disk I/O is in progress, intercept the BIOS interrupts that interacts with the disk devices (both floppy and hard). The intercept functions increment a busy flag, call the original interrupt, and then decrement the busy flag when the BIOS interrupt returns. In Listing 14.1, these functions are *thirteen_handler()*, *twenty_five_handler()*, and *twenty_six_handler()*. All three routines increment and decrement the same flag, *busy_flag*. If the flag is non-zero, one of these routines is in progress. The breakout tool will not terminate the foreground process until *busy_flag* is *0*.

Since DOS itself is generally not reentrant, the breakout tool must determine that *Int 21h* function calls can safely be made before calling the clean-up function and terminating the application. The undocumented but well-publicized *Int 21h* function *34h* returns a pointer to a flag commonly called the *InDOS* flag. If this internal DOS flag is *0*, it is safe to make *Int 21h* function calls. If the flag is non-zero, it is not safe to make DOS function calls. There are exceptions to this rule, but for the purposes of this article, I will ignore them. The function *get_indos_addr()*

(Listing 14.1) uses function `34h` to retrieve the address of the `InDOS` flag and then stores the address in a code segment variable for later use.

The last item in determining machine stability is whether the system is handling a critical error. If a critical error is in progress, the breakout tool should not attempt to terminate the application. DOS maintains a second undocumented flag that, when set to `0`, indicates that a critical error is not in progress. The critical error flag is located one byte after the `InDOS` flag in DOS v2.x, and one byte before the `InDOS` flag in DOS v3+. (In Compaq DOS v3.0, the critical error flag is located `1aah` bytes before the `InDOS` flag.) The function `get_critical_error_addr()` (Listing 14.1) obtains the address of the critical error flag, using the method appropriate for the version of DOS, and stores the address in a code segment variable.

Implementing the Breakout Tool

The breakout tool must determine whether it should break out of the program when its hot key is pressed. Using *Ctrl-Break* as the hot key saves some coding effort, since `Int 9` is already set up to detect this key combination and sets a flag when it is encountered. Because `Int 9` execution is an asynchronous event triggered by a hardware interrupt, pressing *Ctrl-Break* will always set the BIOS break flag, as long as ints are enabled. Anytime the hot key is pressed, the BIOS break flag is set, indicating that program termination is desired. The breakout tool can then check the BIOS break flag state periodically.

Listing 14.1 contains the remaining routines for the breakout tool. The only externally visible functions are `insure()`, which installs the breakout tool, and `cancel()`, which removes it. The application under development can install and remove the breakout tool as often as necessary. `insure()` checks that the breakout tool isn't already installed before attempting to install it, and `cancel()` makes sure the breakout tool is installed before trying to remove it.

The `insure()` function expects a single parameter, a `far` pointer to the function, to be called when *Ctrl-Break* is pressed. This clean-up function may perform any tasks necessary, such as restoring the video mode or restoring interrupt vectors captured by the application before the application is terminated. If your application doesn't require a clean-up function, simply pass the `NULL` pointer as the parameter.

Before proceeding with the installation process, `insure()` inspects the local variable `insured_psp`. If `insured_psp` is non-zero, the breakout tool is already installed, and `insure()` returns `1` to indicate that the requested insurance wasn't put in place. If insurance isn't already in place, `insure()` checks the DOS version. Since this version of the breakout tool requires DOS v3.0 or higher to establish the current Program Segment Prefix (PSP), `insure()` verifies the DOS version as 3.0 or higher before continuing with the installation. The function then retrieves the segment address of the PSP for the application installing the breakout tool and stores it in the local variable `insured_psp`. Later, when the hot key is detected, this address must match the PSP address of the currently executing program before the breakout tool will terminate the program. By requiring the currently executing program's PSP to match the PSP of the installer, you guarantee that you are not about to terminate some other application by mistake. Installation continues by storing the address of the clean-up function in a local variable, obtaining the addresses of the `InDOS` and `critter` flags, saving the application `SS` and `SP`, making sure the BIOS break flag is clear, and installing the interrupt intercept routines. Saving the application stack permits a small memory model application to use certain C library functions that, for some reason, can't handle a different stack. The large memory model avoids this stack manipulation. `insure()` returns `0` to indicate successful installation.

At the heart of the breakout tool is the `Int 8` intercept routine, `int_8h_handler()`. At every timer tick, `int_8h_handler()` calls the original `Int 8` routine to allow the interrupt to be serviced first. Since the original `Int 8` routine issues the End Of Interrupt (EOI) before it returns control, the intercept code doesn't have to. `int_8h_handler()` checks the `in_already` flag before entering the post `Int 8` code to avoid re-entering the post-processing block on subsequent timer ticks. If the `in_already` flag is non-zero, control is immediately returned to the process that was interrupted by the timer tick. If `in_already` is `0`, the code continues by setting the `in_already` flag and switching to an internal stack before saving the necessary registers used by the process. If the BIOS break flag is set and the `InDOS` and `critter` flags are clear, the `cancel()` function is called to remove the captured interrupts. After these tasks are accomplished, the stack is switched to the stack saved during the installation process, and control is transferred to the clean-up function. The clean-up function is not expected to return.

If you write the clean-up function in C, as in Listing 14.2, you must declare it as a `far` function and use the special modifier, `interrupt`. Both Microsoft and Turbo C provide the `interrupt` modifier, designed to permit writing interrupt service routines in C. The `interrupt` modifier causes the compiler to generate special entry code for the function that saves all registers and insures that the `DS` register contains the address of the `DGROUP` segment. Special function exit code is also generated that restores the entry registers and returns using an IRET. When the clean-up function has finished its tasks, the C library function `exit()` terminates the application.

Listing 14.2 also contains a sample driver to test the breakout tool. The only thing to note about the driver is that the assembly language function `cancel()` is registered with the C library function `atexit()`. Passing the function pointer to `atexit()` registers the function to be called during normal termination. Registering `cancel()` with `atexit()` insures that when the application terminates normally, the `cancel()` function is called before returning to DOS. All the interrupt vectors intercepted by the breakout tool are thus restored before the application terminates.

Many similarities exist between this breakout tool and a "pop-up" TSR. Both require a means of detecting the hot key pressed, and both must be able to interrupt the foreground process, but must do so only when the operating environment is in a stable state. Because of these similar requirements, many of the techniques used here are identical to those used to write successful "pop-up" TSR programs.

Though none of us intentionally codes infinite loops, they happen more often than most of us would like to admit. By adding the breakout tool to your application during development, you can eliminate the frustration of having to reboot. Code which hangs the machine can be tough to locate and correct. The breakout tool can ease the pain and help speed up the development effort.

Bibliography

Hogan, T., 1988, *The PC Programmer's Sourcebook*, Microsoft Press, Redmond, WA.

Schulman, et al., 1990, *Undocumented DOS*, Addison-Wesley, Reading, MA.

Angermeyer, et al., 1989, The Waite Group's *MS-DOS Developers Guide*, 2nd edition, Howard W. Sams & Co., Indianapolis, IN.

Listing 14.1 `breakout.asm`

```
COMMENT ~
------------------------------------------------------------
 This module contains routines necessary to enable
 Ctrl-Break to "break out" of infinite loops.  A user
 supplied clean up routine may be specified if necessary.

 MASM 5.1/TASM 2.0 - C callable installation &
 de-installation functions.

 Language & memory model independant.  Change the .model
 directive as needed.

 Author:  David Burki
------------------------------------------------------------
END OF COMMENT ~

title BREAKOUT.ASM
page 57,132

% .model m_model, lang

; ----  MACROS and EQUATES  ----

;macro to simulate an interrupt so the interrupt returns
;to the instruction immediately following the call
sim_int macro  num
        pushf
        call    cs:orig_&num
endm

;macro to push selected registers
apush macro   a,b,c,d,e,f,g,h
   irp x,<a,b,c,d,e,f,g,h>
      ifnb <x>
         push    x
      endif
   endm
endm

;macro to pop selected registers
apop macro   a,b,c,d,e,f,g,h
   irp x,<h,g,f,e,d,c,b,a>
      ifnb <x>
         pop     x
      endif
   endm
endm
```

Listing 14.1 *continued*

```
;macro to install a replacement vector
;  --  assumes that ds = code seg of replacing function
;  --  all var names used to save original vector must be
;        2 characters long plus a training "h" (i.e. for
;        int 8h "orig_08h")
install_vector macro vector_num,function_name
        mov     ax,35&vector_num
        int     21h
        mov     word ptr orig_&vector_num,bx
        mov     word ptr orig_&vector_num+2,es
        lea     dx,function_name
        mov     ax,25&vector_num
        int     21h
endm

;macro to un-install a replaced vector
;  --  all var names used to save original vector must be
;        2 characters long plus a training "h" (i.e. for
;        int 8h "orig_08h")
restore_vector macro replaced_vector
        lds     dx,cs:orig_&replaced_vector
        mov     ax,25&replaced_vector
        int     21h
endm

; segment and offset of the BIOS break flag
BIOS_SEG        equ     40h
BREAK_FLAG_OFF  equ     71h

; ----  CODE  ----
.code

; declare externally visible functions
public  insure, cancel

; NOTE:  all variables used here are part of the code
;        segment so the interrupt routines can have easy
;        access to them

; --- storage for the address of the InDOS flag
indos_addr              dd      0
```

Listing 14.1 *continued*

```
; --- storage for the address of the critical error flag
critter_addr            dd      0

; --- storage for the version of dos
dos_major_version       db      0
dos_minor_version       db      0

; --- flag indicating disk i/o in progress
busy_flag               db      0

; --- far pointer to clean up routine
exit_routine            dd      0

; --- storage for original addresses of intercepted
;     interrupt vectors.
orig_08h                dd      0
orig_1bh                dd      0
orig_13h                dd      0
orig_25h                dd      0
orig_26h                dd      0

; --- storage for the segment addr of the psp of process
;     that asked for insurance  -  if zero, insurance isn't
;     in effect
insured_psp     dw      0

; --- flag if still processing in int 8h
in_already      db      0

; --- interrupted program's SS & SP
save_ss         dw      0
save_sp         dw      0

; --- original C stack segment
installer_ss    dw      0
installer_sp    dw      0

; --- local stack while in int_8h_handler
my_stack        dw      256 dup(0)
stack_top       label word

assume ds:@curseg, es:nothing
```

Listing 14.1 *continued*

```
;----------------------------------------------------------
;  GET_DOS_VERSION()
; Obtain the version & revision of DOS.  Store the version
; in the variable "dos_major_version" and the revision in
; the variable "dos_minor_version".  Both variables are code
; segment variables.
;
; Returns:
;       AL - dos minor version number
;       AH - dos major version number
;----------------------------------------------------------
public get_dos_version
get_dos_version proc
        mov     ah,30h
        int     21h
        mov     cs:dos_major_version,al
        mov     cs:dos_minor_version,ah
        ret
get_dos_version endp

;----------------------------------------------------------
;  GET_CRITICAL_ERROR_ADDR()
; This function obtains & saves the address of the
; critical error flag.  For DOS versions 3+, the critical
; error flag is the byte just before the InDOS flag. For
; DOS 2.x, it's the byte just after.
;
; NOTE:  get_dos_version() must be called before this
;        function
;
; NOTE:  For COMPAQ DOS version 3.x, the critical error
;        flag is located 01aah bytes BEFORE the InDOS flag.
;
; Trashes: AX
;----------------------------------------------------------
public get_critical_error_addr
get_critical_error_addr proc    USES ES BX
; --- get the address of the InDOS flag
        mov     ah,34h
        int     21h

; --- dos 3.x or better use byte before InDOS
        cmp     byte ptr cs:dos_major_version,3
        jge     byte_before  ;good_version

; --- dos 2.x, use byte after InDOS flag
        inc     bx
        jmp     store_critter_addr
```

Listing 14.1 *continued*

```
byte_before:
        dec     bx
store_critter_addr:
; --- save the address & return success
        mov     word ptr cs:critter_addr,bx
        mov     word ptr cs:critter_addr+2,es

        ret
get_critical_error_addr endp

;-------------------------------------------------------
;  GET_INDOS_ADDR()
; Uses the undocumented (but well known) function 34h of
; INT 21h to obtain a far pointer to the "InDOS" flag.
; Stores the address of the flag in the code segment
; variable "indos_addr".
;
; Trashes AX
;-------------------------------------------------------
public get_indos_addr
get_indos_addr proc     USES ES BX
; --- get the address
        mov     ah,34h
        int     21h

; --- save address in cs referenced variable
        mov     word ptr cs:indos_addr,bx
        mov     word ptr cs:indos_addr+2,es
        ret
get_indos_addr endp

;-------------------------------------------------------
;  CHECK_CRITTER_FLAG()
; Examine the undocumented "critical error" flag to
; determine if a critical error is in progress.
;
; Returns:
;   carry clear - A critical error is NOT in progress
;   carry set   - A critical error is in progress
;-------------------------------------------------------
public check_critter_flag
check_critter_flag proc USES AX DS SI
; --- ds:si <- address of critter flag
        lds     si,dword ptr cs:critter_addr

; --- check critter flag = zero
        lodsb
        or      al,al
        jz      no_critter
```

Listing 14.1 *continued*

```
; --- critter flag not zero, return carry set
        stc
        jmp     critter_exit

; --- critter flag zero, return carry clear
no_critter:
        clc

critter_exit:
        ret
check_critter_flag endp

;-----------------------------------------------------------
;  CHECK_INDOS_FLAG()
; Examine the undocumented "InDOS" flag to determine if
; non-rentrant DOS functions are currently executing.
;
; Returns:
;     carry clear - InDOS flag is zero.
;     carry set   - InDOS flag is non-zero.
;-----------------------------------------------------------
public check_indos_flag
check_indos_flag proc  USES AX DS SI
; --- ds:si <-- addr of InDOS flag, then load al with
;     the byte at that address
        lds     si,dword ptr cs:indos_addr
        lodsb

; --- indos flag equal zero, DOS is stable
        or      al,al
        jz      dos_stable

; --- no, set carry & return
        stc
        jmp     indos_exit

; --- yes, clear carry and return
dos_stable:
        clc

indos_exit:
        ret
check_indos_flag endp
```

Listing 14.1 *continued*

```
;-------------------------------------------------------------
;  CHECK_BREAK_BIT()
;    This function examines the BIOS Break Flag (bit 7 of
;    the byte at 40:71).
;
; Returns:
;     carry clear - Break Flag is clear.
;     carry set   - Break Flag is set.
;-------------------------------------------------------------
check_break_bit proc    USES DS ES BX
; --- establish addressing to the flag
        mov     bx,BIOS_SEG
        mov     es,bx
        mov     bx,BREAK_FLAG_OFF

; --- is bit 7 set
        test    es:byte ptr [bx],10000000b
        jnz     break_bit_set

; --- no, return with carry clear
        clc
        jmp     check_break_exit

; --- yes, clear it & return carry set
break_bit_set:
        and     es:byte ptr [bx],01111111b
        stc

check_break_exit:
        ret
check_break_bit endp

;-------------------------------------------------------------
; INSTALL_INTERCEPTS()
;    Retrieve & save the current vectors, replacing the
;    vector table entries with the addresses of the
;    intercept routines.
;
; Entry:
;    DS = CS
; Returns:
;    No value
; Trashes:
;    AX, BX, DX, ES
;-------------------------------------------------------------
install_intercepts proc
assume ds:@curseg
; --- capture int 8h vector
        install_vector 08h,int_8h_handler
```

Listing 14.1 *continued*

```
; --- capture int 13h vector
        install_vector 13h,thirteen_handler

; --- capture int 25h vector
        install_vector 25h,twenty_five_handler

; --- capture int 26h vector
        install_vector 26h,twenty_six_handler

; --- capture int 1bh vector
        install_vector 1bh,break_handler

        ret
install_intercepts endp

;--------------------------------------------------------------
; INSURE( cleanup_function )
;   void (*PFV)();  /* far pointer to cleanup function */
;
;   Installs the breakout tool.
;
; Returns:
;       AX = zero     -  installed successfully
;       AX not zero  -  did not install
;--------------------------------------------------------------
insure proc          USES DS ES SI DI, CLEAN_UP:FAR PTR
; --- force ds = cs
        push    cs
        pop     ds
assume ds:@curseg

; --- currently in palce?
        cmp     word ptr insured_psp,0

; --- yes, return to caller with no action
        jnz     not_installed

; --- get dos version
        call    get_dos_version

; --- if not dos 3.x don't install
        cmp     ah,3
        jl      not_installed

; --- retrieve & save the psp of the current process
; --- function 62h (get PSP) is DOS 3+ only
        mov     ah,62h
        int     21h
        mov     insured_psp,bx
```

Listing 14.1 *continued*

```
; --- get the addr of clean up function & save it
        les     bx,CLEAN_UP
        mov     word ptr exit_routine+2,es
        mov     word ptr exit_routine,bx

; --- get the addr of "InDOS" flag
        call    get_indos_addr

; --- get the gritical error flag address
        call    get_critical_error_addr

; --- install the interrupt intercepts
        call    install_intercepts

; --- insure BIOS break flag is clear
        mov     bx,BIOS_SEG
        mov     es,bx
        mov     bx,BREAK_FLAG_OFF
        and     es:byte ptr [bx],01111111b

; --- save installer's SS and SP
        mov     installer_ss,ss
        mov     installer_sp,sp

; --- set good return code & clear "in already" flag
;     for int 8h handler
        xor     ax,ax
        mov     byte ptr in_already,al
        jmp     insure_exit

not_installed:
        mov     ax,1

insure_exit:
        ret
insure endp

assume ds:nothing
```

Listing 14.1 *continued*

```
;----------------------------------------------------------
; CANCEL()
;   Restore the original vectors and clear the
;   "insured_psp" address.
; Returns:
;   No value
; Trashes:
;   AX
;----------------------------------------------------------
cancel proc             USES DS DX
; --- if not installed, get out with no action
        mov     dx,cs:insured_psp
        or      dx,dx
        jz      no_cancellation

; --- get the original int 8h  & restore it
        restore_vector 08h

; --- get the original int 1bh & restore it
        restore_vector 1bh

; --- get the original int 13h & restore it
        restore_vector 13h

; --- get the original int 25h & restore it
        restore_vector 25h

; --- get the original int 26h & restore it
        restore_vector 26h

; --- clear the insured_psp address
        mov     cs:word ptr insured_psp,0

no_cancellation:
        ret
cancel endp
```

Listing 14.1 *continued*

```
assume cs:@curseg
;-----------------------------------------------------------
;  INT_8H_HANDLER()
;   Intercept for timer tick interrupt.
;   Checks the BIOS break flag (bit 7 at 40:71) on each
;   invocation.
;   Flagged to prevent rentry.
;-----------------------------------------------------------
int_8h_handler proc far
; --- let int 8 do it's stuff
        sim_int 08h
        cli

; --- if int 8 post processing is still going on, don't
;     re-enter - return to the interrupted code
        cmp     cs:byte ptr in_already,1
        jnz     do_break_ck
        iret

do_break_ck:
; --- flag that we're processing an int 8h & swap stacks
        mov     cs:byte ptr in_already,1
        mov     cs:save_ss,ss
        mov     cs:save_sp,sp
        push    cs
        pop     ss
        mov     sp,offset cs:stack_top
        sti
        apush   ax,bx,cx,dx,ds,es,si,di

; --- if the BIOS break flag is NOT set, iret to caller
        call    check_break_bit
        jnc     no_break

; --- if disk i/o in progress, iret back to caller
        cmp     cs:busy_flag,0
        jnz     no_break
```

Listing 14.1 *continued*

```
; --- if not in dos or critical error
        call    check_indos_flag
        jc      no_break
        call    check_critter_flag
        jc      no_break

; --- if the psp of the caller isn't the same as
;     the psp when installed, skip it
        mov     ah,62h
        int     21h
        cmp     bx,cs:insured_psp
        jnz     no_break

; --- BIOS break flag was set, everything is stable and the
;     active PSP is the same as the one that installed the
;     insurance.  clear the BIOS break bit and uninstall
;     the intercepted vectors
        call    cancel

; --- if a user supplied exit routine is installed, do it
        mov     bx,word ptr exit_routine+2
        or      bx,bx
        jnz     do_user_supplied_exit

; --- otherwise, DOS terminate function - error level = 1
        mov     ax,4c01h
        int     21h

do_user_supplied_exit:
; --- get back BX & ES plus clean off CS, IP & FLAGS pushed
;     on original interrupt, then jump to user exit routine
        mov     ss,cs:installer_ss
        mov     sp,cs:installer_sp
        jmp     cs:exit_routine

no_break:
; --- clean up the stack
        apop    ax,bx,cx,dx,ds,es,si,di
        cli
        mov     ss,cs:save_ss
        mov     sp,cs:save_sp
        sti
        mov     cs:byte ptr in_already,0

skip_it:
; --- iret to the int 8 caller
        iret

int_8h_handler endp
```

Listing 14.1 *continued*

```
;---------------------------------------------------------
;  BREAK_HANDLER()
;   Simple replacement for int 1bh - no more than an iret
;   to keep the "^C" off the display screen.
;---------------------------------------------------------
break_handler proc far
        iret
break_handler endp

;---------------------------------------------------------
;  THIRTEEN_HANDLER()
;   intercept for int 13h
;   Increments the busy flag on entry, calls the original
;   interrupt service routine, and decrements the busy flag
;   when the disk routine is complete. The busy flag must
;   be zero to indicate no disk i/o in progress.
;---------------------------------------------------------
thirteen_handler proc far
        pushf
        inc     cs:busy_flag
        popf
        sim_int 13h
        pushf
        dec     cs:busy_flag
        popf
        iret
thirteen_handler endp

;---------------------------------------------------------
;  TWENTY_FIVE_HANDLER()
;   intercept for int 25h
;   Increments the busy flag on entry, calls the original
;   interrupt service routine, and decrements the busy flag
;   when the disk routine is complete. The busy flag must
;   be zero to indicate no disk i/o in progress.
;---------------------------------------------------------
twenty_five_handler proc far
        pushf
        inc     cs:busy_flag
        popf
        sim_int 25h
        pushf
        dec     cs:busy_flag
        popf
        iret
twenty_five_handler endp
```

Listing 14.1 *continued*

```
;---------------------------------------------------------
;  TWENTY_SIX_HANDLER()
;   intercept for int 26h
;   Increments the busy flag on entry, calls the original
;   interrupt service routine, and decrements the busy flag
;   when the disk routine is complete. The busy flag must
;   be zero to indicate no disk i/o in progress.
;---------------------------------------------------------
twenty_six_handler proc far
        pushf
        inc     cs:busy_flag
        popf
        sim_int 26h
        pushf
        dec     cs:busy_flag
        popf
        iret
twenty_six_handler endp

end

; End of File
```

Listing 14.2 `testbrk.c`

```
/* ---------------------------------------------------------
    File:  testbrk.c

    This is the test driver for the routines in breakout.asm.

    Author:  David Burki
   --------------------------------------------------------- */
#include <stdio.h>

// define far a pointer to a void function
typedef void (far *PFV)();

// function prototypes
void insure(PFV);
void interrupt far pre_termination(void);
void cancel(void);

/* -------------------------------------------------------
   function which performs any necessary clean-up before
   terminating the application
   ------------------------------------------------------- */
void interrupt far pre_termination()
{
  printf("\nIn the pre-termination function\npress any key\n");
  getch();
  exit(1);
}

// test driver main function
main()
{
// register the cancel function so exit will call it
  atexit(cancel);

// install the breakout tool
  insure((PFV)pre_termination);

// a never ending loop
  for(;;)
     ;
}

/* End of File */
```

Listing 14.3 `testbrk.exe` *makefile.*

```
## -------------------------------------------------------------
## This is the make file to build the executable test
## driver for the breakout tool.
##
## Author:  David Burki
## -------------------------------------------------------------

AMODEL= large                   ## define the memory ...
CMODEL= l                       ## model for the project
LANGUAGE= C                     ## the high level language
TC_LIBDIR= E:\TC\LIB            ## and the library directory(s)

testbrk.exe:    listing1.obj listing2.obj
        tlink $(TC_LIBDIR)\C0$(CMODEL).obj listing2 \
              listing1, testbrk.exe, nul, \
              $(TC_LIBDIR)\C$(CMODEL)

listing1.obj:   listing1.asm
        tasm -Dm_model=$(AMODEL) -Dlang=$(LANGUAGE) $*;

listing2.obj:    listing2.c
        tcc -c -m$(CMODEL) $*.c

## End of File
```

Chapter 15

Checking for Debug Information

Matt Pietrek

In the last-minute rush to get a product out the door, mistakes get made. In my experience, one of the most common and embarrassing of these is accidentally shipping debugging information with your product. In this chapter, I present a utility, `dbginfo.exe`, that can help you quickly check for this situation, and thus avoid this particular embarrassment.

Inadvertently supplying debugging information with your program is undesirable for several reasons. First, debugging information is a fairly complete representation of your program, in that it contains your function and variable names, as well as the formats of your data structures. Someone familiar with the format of your debug information can use it to greatly speed up the process of reverse engineering. For instance, much of the information in Addison-Wesley's *Undocumented Windows* was obtained by examining the CodeView information supplied with the debugging DLLs in the Microsoft Windows SDK (for more on the format of the debug information, see the sidebar).

Second, debug information can significantly add to the file size of the `.exe` or DLL. In my experience with Borland and Microsoft debugging formats, they typically double the size of the file for a non-trivial program. When you are trying to squeeze your product into the fewest possible disks for distribution, debug information can be very significant dead weight.

Last, leaving debug information in a shipping program just plain looks unprofessional. If the programmer did not take the time to implement proper build procedures, are you going to have confidence in the code itself?

A related issue arises if your product consists of or includes `.lib` or `.obj` files. Debug information is produced by the compiler and stored in `.obj` files, along with the code and data segments. During the link phase, debug information from the various `.obj` files is combined to form the debug information in the executable file. Although `dbginfo` does not

Looking at the Debug Info

The two most common formats for debugging information on the PC are those used for Borland's Turbo Debugger and Microsoft CodeView. The format for the debug info in Borland `.obj` and executable files can be found in the Borland *Open Architecture Handbook*. The Microsoft CodeView executable format is described in the *Microsoft C Developer's Toolkit Reference* (Microsoft, 1990, Part No. 18161).

If you want to examine the debug information in a particular executable file, `@5ITDUMP.EXE` (a tool written by the author and included with the Borland compilers) can display both Turbo Debugger and CodeView style debug information. In addition, `TDUMP` can display the Borland debug information that's included in `.obj/.lib` files.

If the sheer volume of information from `TDUMP` overwhelms you, you can narrow down the amount of information provided by using some command line switches.

With an executable file, to skip over the display of the executable information and only display the debug info, use the `-ex` switch. To see check for debug information in `.obj` or `.lib` files, the same caveats described earlier apply. In addition, the Borland C/C++ compilers by default produce "autodependency" information that can also add to the size of your `.obj` and `.lib` files. If you intend to ship your `.obj` or `.lib` files, you can use Borland's `-X` option to suppress generation of this information. The sidebar describes how to examine `.obj` and `.lib` files in more detail.

Finding CodeView1 Debug Information

Borland and Microsoft format debug information differently, so you have to perform two checks. Luckily, in both cases the debugging information can be found at the end of the executable file.

the Borland debug information in its "raw" tabular format, use the `-v` switch. For example:

```
TDUMP -ex -v myfile.exe
```

For Borland `.obj` files,

```
TDUMP -oicoment filename.obj
```

will eliminate much of the information not related to debug information and autodependencies. For `.obj` files with CodeView information, try

```
TDUMP -oisegdef filename.obj
```

If you see segments that have names starting with "`$$`" (for instance "`$$TYPES`"), then the `.obj` file contains CodeView style information.

As Figure 15.1 shows, the Microsoft CodeView debugging information is relatively easy to find. For DOS, Windows, and OS/2 executables, the last eight bytes of the CodeView information contain three things:

- A 2-byte "NB" signature.
- A 2-byte version number.
- The length of the debug information (an `unsigned long`).

To check for CodeView information, first read the last eight bytes of the file to verify that it starts with "NB." If that signature exists, then read the beginning of the debug information and verify that it also starts with "NB." To find the start of the debug information, seek backwards from

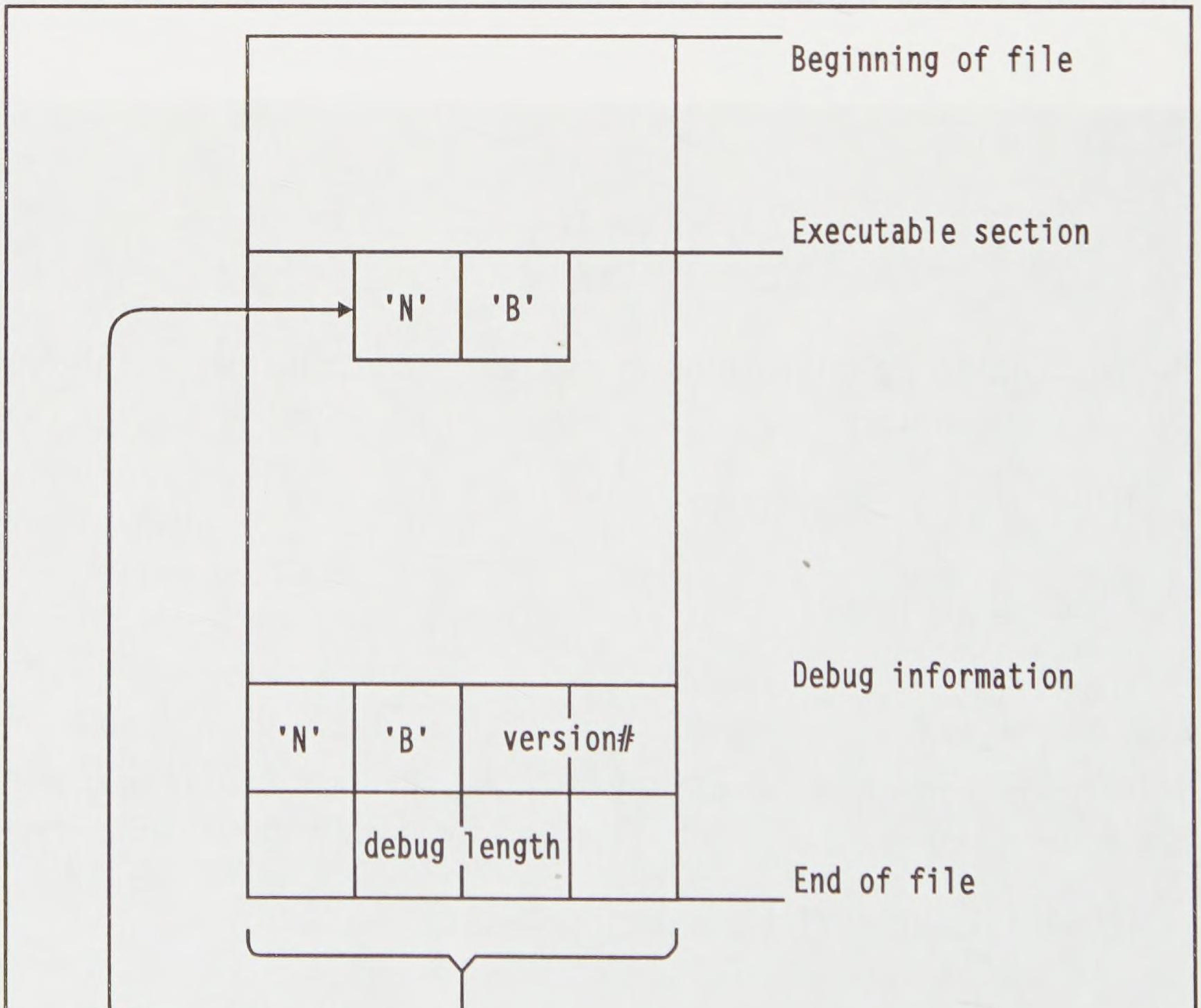

Beginning of file
Executable section
'N'
'B'
Debug information
'N'
'B'
version#
debug length
End of file

Figure 15.1 *Format of CodeView executables.*

the end of the file by the number of bytes given in the length field in the last four bytes of the file.

Finding Turbo Debugger Debug Information

The check for CodeView debugging information is the same whether you are examining a DOS executable or a Windows executable. Borland's debugging information, however, gets stored differently in Windows executables than in DOS executables.

When producing a Windows executable, both `TLINK` and Turbo Pascal for Windows produce a CodeView "shell" around the Turbo Debugger information. Because of this, the previous algorithm for detecting CodeView debugging information will work the same for a Borland Windows executable that contains debugging information. You should note, however, that the debugging information between the two "NB" signatures is not CodeView information, but is, instead, Turbo Debugger information. This shell around the Turbo Debugger information is necessary because Microsoft's resource compiler (`rc.exe`) discards anything at the end of an executable file that it does not recognize as CodeView information. The actual Turbo Debugger information starts 16 bytes past the first "NB" signature in the file. The start of the Turbo Debugger information can be identified by the two-byte sequence: `FBh`, `52h`.

Although a single check suffices to detect debugging information in Microsoft DOS and Windows executables and in Borland Windows executables, you need a different algorithm to detect debugging information in Borland DOS executables. The algorithm is to find the end of the executable portion of the file, and then look for the `FBh`, `52h` signature. For a non-overlaid DOS executable, the end of the file is found by using the fields in the "Old EXE" header. The format of the header is given in `oldexe.h` (Listing 15.1) and is described in a number of books, including the Microsoft's *MS-DOS Encyclopedia*. If the executable contains overlays, you have to first find the end of the "DOS executable" portion of the file, and then skip over the overlay information section before looking for the debug information signature. The format of the Borland `VROOM` overlays section is given in Borland's *Open Architecture Handbook*.

The `dbginfo` *Code*

The code for *dbginfo* is in *oldexe.h* (Listing 15.1) and *dbginfo.c* (Listing 15.2). My goal was to let you quickly search for debug information in any set of executable files. You might also find it handy to use in your makefile when doing a "production build." With a little extra work, you could modify it to remove the debug information from the end of the file, à la Borland's *TDSTRIP*.

A couple of sections of the code in *dbginfo.c* are worth looking at in a little detail. First, the *IterateThroughFiles()* function allows you to specify any legal file specification on the command line, including wildcards. For instance, you could type:

```
dbginfo c:\myproj\output\*.*
```

The wildcards are expanded by using *_dos_findfirst()* and *_dos_findnext()*. *dbginfo* will display the complete file and pathname of any executables that contain either Borland or Microsoft style debug information.

To allow you to use wildcards but not slow down the search, the *CheckForDebugInfo()* function first verifies that the file passed to it is an executable file. Because all executable files contain an "Old EXE" header, the potential "Old EXE" header is read in, and the first two bytes are checked for the "MZ" signature. If the file is not an executable file of some sort, the function returns immediately.

Of the four most common permutations of executables and debug formats (Borland/Microsoft, DOS/Windows), three can be found by looking for the CodeView signatures. Naturally, then, *dbginfo* first looks for debug information containing the CodeView signatures. If it does not find the CodeView signatures, it checks for the *1* remaining case (a Borland DOS executable).

The *HasTDStyleInfo()* function examines the "Old EXE" header to see if the file is really an OS/2 or Windows executable. If so, it returns immediately, since the debug information for those types of files would already have been found in the *HasCodeViewStyleInfo()* function. The one tricky thing that *HasTDStyleInfo()* does is skip over a *VROOM* overlay section if one is present. At several points, *HasTDStyleInfo()* will return *FALSE* if the end of the executable section coincides with the end of the entire file.

Summary

Testing for debugging information before shipping a program should be mandatory procedure for every developer: `dbginfo` allows you to do this quickly and easily. If you apply the techniques described here, you will find yourself in a much better position to avoid the pitfalls of a hasty rollout.

References

Ray Duncan, ed. 1988, *The MS-DOS Encyclopedia.* Microsoft Press, Redmond.

Open Architecture Handbook: The Borland Developer's Technical Guide, 1991, Borland.

Andrew Schulman, David Maxey, and Matt Pietrek, 1992, *Undocumented Windows.* Addison-Wesley Publishing Co.

Listing 15.1 OLDEXE.H

```
//==================================
//  DBGINFO by Matt Pietrek, 1992
//==================================
typedef struct
{          /* DOS - 'old' EXE Header */
    unsigned char   signature[2];   /* EXE File validity mark    */
    unsigned short  lastsector;     /* length MOD 512            */
    unsigned short  numblocks;      /* number of whole 512 bytes */
    unsigned short  reloc_size;     /* Relocation table size     */
    unsigned short  header_size;    /* Size of header record     */
    unsigned short  min_mem;        /* Minimum Load Memory       */
    unsigned short  max_mem;        /* Maximum Load Memory       */
    unsigned short  stack_seg;      /* Stack segment segment     */
    unsigned short  stack_off;      /* Stack segment offset      */
    unsigned short  checksum;       /* File load checksum        */
    unsigned short  code_off;       /* Entry point offset        */
    unsigned short  code_seg;       /* Entry point segment       */
    unsigned short  reloc_offset;   /* Offset to relocation table*/
    unsigned short  overlay_number; /* Overlay Number            */
} OLDEXEHEADER;

/* End of File */
```

Listing 15.2 DBGINFO.C

```
//=================================
//  DBGINFO by Matt Pietrek, 1992
//=================================
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <dir.h>
#include "oldexe.h"

#define CV_SIGNATURE                0x424E  /* 'NB'   */
#define TD_SIGNATURE                0x52FB
#define BOR_OVERLAY_SIGNATURE1      0x4246L /* 'FB' */
#define BOR_OVERLAY_SIGNATURE2      0x564FL /* 'OV' */

char HelpString[] =
"DBGINFO Version 1.0  Copyright (c) 1992 Matt Pietrek\n\n"
"Syntax: DBGINFO <filespec>\n\n"
"Lists all files for a given file specification that contain Borland\n"
"Turbo Debugger or Microsoft CodeView information.\n";

/*--------------------------------------------------------------*/
/* Returns 1 if the file contains the 'NB' signatures used by   */
/* CV, CVW, and TDW, 0 if not                                   */
/*--------------------------------------------------------------*/
int HasCodeViewStyleInfo(FILE *exefile)
{
    unsigned short signature=0;
    unsigned long debugInfoLen;

    /* Seek 8 bytes from the end of the file, */
    /* and look for the 'NB' signature        */

    fseek(exefile, -8, SEEK_END);
    fread(&signature, sizeof(signature), 1, exefile);

    if ( signature != CV_SIGNATURE )
        return 0;

    /* Read in the length of the debug information */

    fseek(exefile, -4, SEEK_END);
    fread(&debugInfoLen, sizeof(debugInfoLen), 1, exefile);

    /* Seek to the start of the debug info, and  */
    /* look for the corresponding 'NB' signature */

    signature = 0;
    fseek(exefile, 0-debugInfoLen, SEEK_END);
    fread(&signature, sizeof(signature), 1, exefile);

    return (signature == CV_SIGNATURE) ? 1 : 0;
}
```

Listing 15.2 *continued*

```
/*--------------------------------------------------------------*/
/* Returns 1 if the file contains debug information for the DOS */
/* version of Turbo Debugger, 0 otherwise.                      */
/*--------------------------------------------------------------*/
int HasTDStyleInfo(FILE *exefile, OLDEXEHEADER *oldheader)
{
    unsigned long endOfExeImage, fileLen, overlayLen;
    unsigned short signature=0;

    /* Return failure if the file is a Windows or OS/2 file */

    if ( oldheader->reloc_offset == 0x40 )
        return 0;

    /* find out how long the file is */

    fseek(exefile, 0, SEEK_END);
    fileLen = ftell(exefile);

    /* Calculate how much of the file the */
    /* DOS Loader will pay attention to.  */

    endOfExeImage = oldheader->numblocks * 512L;
    if ( oldheader->lastsector )
        endOfExeImage -= (512 - oldheader->lastsector);

    if ( endOfExeImage >= fileLen )
        return 0;

    /* seek to the end of the Exe image portion, */
    /* and read in the signature word .          */

    fseek(exefile, endOfExeImage, SEEK_SET);
    fread(&signature, sizeof(signature), 1, exefile);

    /* Is this a Borland overlay file???  If so,   */
    /* we must get past the overlay portion before */
    /* we can look for the TD debug info signature */

    if ( signature == BOR_OVERLAY_SIGNATURE1 )
    {
        fread(&signature, sizeof(signature), 1, exefile);

        if ( signature != BOR_OVERLAY_SIGNATURE2 )
            return 0;

        /* Find out how long the overlay info is */

        fseek(exefile, endOfExeImage+4, SEEK_SET);
        fread(&overlayLen, sizeof(overlayLen), 1, exefile);

        endOfExeImage += (overlayLen+16);
```

Listing 15.2 *continued*

```
        if ( endOfExeImage >= fileLen )
            return 0;

        /* Seek past the end of the overlay info, */
        /* and read in another signature word, in */
        /* preparation for looking for debug info */

        fseek(exefile, endOfExeImage, SEEK_SET);
        fread(&signature, sizeof(signature), 1, exefile);
    }

    return (signature == TD_SIGNATURE) ? 1 : 0;
}

/*----------------------------------------------------------------*/
/* Opens a file, and verifies that it is some kind of .EXE  by    */
/* looking for the 'old' EXE signature.  It then calls the       */
/* above routines to see if the file contains debug info.        */
/*----------------------------------------------------------------*/
void CheckForDebugInfo(char *filename)
{
    FILE *exefile;
    OLDEXEHEADER oldheader;

    if ( !(exefile = fopen(filename, "rb")))
        return;

    /* Read in the 'old' style DOS EXE header, and */
    /* verify that it's an .EXE by looking for 'MZ' */

    fread(&oldheader, sizeof(oldheader), 1, exefile);

    if ( (oldheader.signature[0] != 'M') ||
         (oldheader.signature[1] != 'Z') )
    {
        fclose(exefile);
        return;
    }

    /* If either of the 2 functions return 1,  */
    /* print out the full pathname of the file */

    if ( HasCodeViewStyleInfo(exefile) ||
         HasTDStyleInfo(exefile, &oldheader) )
        printf("%s\n", filename);

    fclose(exefile);
}
```

Listing 15.2 *continued*

```
/*------------------------------------------------------------*/
/* Takes the filespec given on the command line, and uses     */
/* _dos_findfirst/_dos_findnext to iterate through the files.  */
/* For each file, CheckForDebugInfo() is called to look for   */
/* debug information in the file.                             */
/*------------------------------------------------------------*/
void IterateThroughFiles(char *filespec)
{
    char fullPathName[_MAX_PATH];
    char drive[_MAX_DRIVE], dir[_MAX_DIR];
    char file[_MAX_FNAME], ext[_MAX_EXT];
    struct find_t ffblk;
    int done;

    /* Break the passed in filespec into it's component */
    /* parts, so that we can construct a full path name */
    /* for each file found by _dos_findfirst/next.      */

    _fnsplit(filespec, drive, dir, file, ext);
    sprintf(fullPathName,"%s%s%s%s", drive, dir, file, ext);

    /* Begin iterating through all of the files. */

    done = _dos_findfirst(fullPathName, _A_NORMAL, &ffblk);
    while ( !done )
    {
        sprintf(fullPathName,"%s%s%s", drive, dir, ffblk.name);
        CheckForDebugInfo(fullPathName);
        done = _dos_findnext(&ffblk);
    }
}

int main(int argc, char *argv[])
{
    if ( argc !=2 )
    {
        printf(HelpString);
        exit(1);
    }

    IterateThroughFiles(argv[1]);

    return 0;
}

/* End of File */
```

Chapter 16

Programming the PC Game Port

John F. Jarrett

Introduction to the Game Port

The usefulness of the joystick is often downplayed because of the wide acceptance and use of the mouse, but in fact, the uses of the joystick extend far beyond its popular role in video games. A vast number of physically challenged persons, for instance, cannot use a mouse but are nevertheless able to operate a joystick to control a wheelchair or position a PC cursor for a "speech machine." The joystick is an important option in DOS and Windows programming. A mouse-aware program can easily substitute a joystick for *X* and *Y* input and send values to the mouse driver, thus positioning the mouse cursor.

Joystick ports are generally referred to as game ports. They are still widely available and are almost always included in a multi I/O card that also supplies a printer port, a serial port, a floppy disk drive interface, and a hard disk drive interface. Game ports are also provided by the manufacturers of most sound cards because if you change the designation of a couple of unnecessary pins on the game port's `DB15` connector, MIDI (Musical Instrument Digital Interface) input and output becomes available as well.

Most BASIC programmers have all they need to program a joystick in the functions `STICK()` and `STRIG()`. These functions, or similar functions, are supplied with the interpreter or run-time library of most BASIC compilers. If you program in C, however, you will need to know a few basics about the joystick port and how it is addressed.

On each joystick there are two potentiometers (one for the *X* axis; another for the *Y* axis) and two buttons. The stick itself is mechanically attached to the potentiometers so that movement of the stick can be converted to a change in resistance in one or both of the potentiometers. Each potentiometer varies the amount of current passing into a capacitor. The game port can accept input from two separate joysticks for a total of four potentiometers and four buttons. These inputs can also interface with other devices. The joystick port can sample up to 16 button presses if you attach a 16-key matrix via the one-of-16 Binary-Coded Decimal (BCD) encoder Integrated Circuit (IC) to the four button or switch pins of the game port. Then by checking the status of the buttons on the game port, you can determine the status of all four pins.

The joystick's *X* and *Y* position is dependent on the time it takes for the capacitor voltage to change back to its neutral state. It takes time to charge the capacitor, which other circuitry senses. When the capacitor is sufficiently charged, the circuit changes the joystick port value back to its original or neutral state. Therefore, we must measure the time it takes for a specific joystick bit to go from a logical *1*, which occurs after charging (or "firing") the circuit, back to a logical `0`, which is its at-rest or neutral state. The time that this takes gives us the approximate position of the joystick along either its *X* or *Y* axis, depending upon which port bit you are checking. The larger the resistance in the joystick potentiometer, the greater the length of time it takes for the capacitor to charge and the greater the value returned. Since we cannot get the position of the joystick directly, we must determine a relative position using multiple functions. Since there is a potentiometer for each axis, moving the stick all the way to the right, whether the stick is in an upward position or a downward position, results in a higher value being returned for the *X* axis. Positioning the stick all the way down, whether the stick is positioned to the left or right, results in a higher returned value for the *Y* axis. The largest *X* and *Y* values will occur when the stick is positioned in the lower right *corner* of the joystick. The smallest values occur when the stick is positioned in the upper left *corner* or position of the joystick. With the stick in its middle or default centered position, the return values should be about half what the maximum return values will be. Downward or rightward movement of the stick increases the potentiometer resistance, which increases the time for the capacitor to charge and equally increases the value returned. By checking the *X*-axis first and then the *Y*-axis, you can determine the relative position of the joystick. This type of resistor/capacitor (R/C) timing network is also called a "one shot" because it executes only one R/C timing event or pulse. Charging a one shot R/C circuit is also known as "firing" the one shot.

The game port is located at address `201H` (hex). All values of each joystick and each button are returned in one byte at this address. The values for the joystick axes are contained in the lower four bits of the byte returned after firing the one shot. The button status is returned in the upper four bits of the same byte. Table 16.1 shows each bit of the port at address `201H` and its status.

The game port can be accessed directly with a port instruction to set the port's initial value (such as when firing the game port's one shot) or to fetch the port's value. The game port can also be addressed through BIOS services with an AT compatible computer whose BIOS supports the game port. This method is discussed later in this text.

Bit	**Status**
0	one shot for joystick A, X axis
1	one shot for joystick A, Y axis
2	one shot for joystick B, X axis
3	one shot for joystick B, Y axis
4	button 1 for joystick A
5	button 2 for joystick A
6	button 1 for joystick B
7	button 2 for joystick B

Table 16.1 ***The bits of the game port.***

The following functions use the direct access method. The port functions fire the one shot pulse, making the bits climb to a logical *1*, and then continuously check the appropriate bit for its return to a logical *0*.

To simplify our discussion, I will concentrate on the primary joystick, joystick A. The same events and principles apply to joystick B equally. When the bit returns to logical *0*, the function is then exited. The time, or number, is actually the number of iterations or loops the function performs before the bit returns to a logical *0*. This gives us the approximate position of the joystick. The higher the value, the greater the resistance of the potentiometer, and the stick will usually be in its bottommost or rightmost position. The lower the value, the lower the resistance of the potentiometer, and the stick will be in its uppermost or leftmost position. Figure 16.1 graphically shows this relationship.

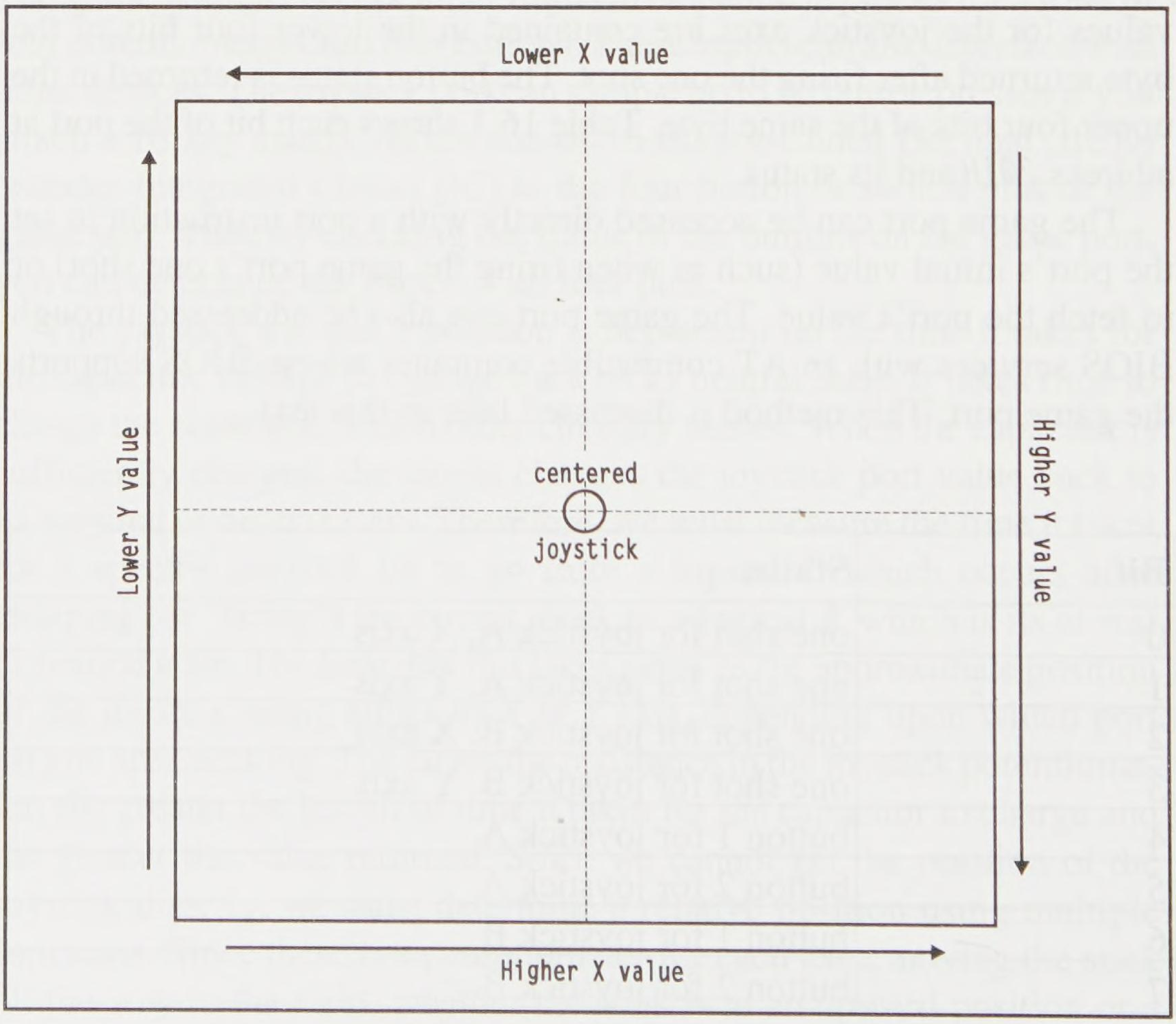


Figure 16.1 *Joystick position vs. relative X and Y return values.*

The game port is generally a *DB15* type 15-pin female connector. Figure 16.2 shows the game port pinouts.

Note that when a button is pressed, +5 volts (a logical *1*) is applied to the pin. The joystick potentiometers are connected to +5 volts and the pins are connected to the wipers or center conductor of the potentiometers.

The Joystick X and Y Axes

The first thing you must do is "fire" the one shot and begin counting how long it takes for the joystick position to return to a logical *0*. Remember that "firing" the one shot makes the bit, either bit *0*, *1*, *2*, or *3*, go to a logical *1* or high state (+5 volts). The R/C network is actively charging a capacitor on the game card. When the capacitor is charged sufficiently, the bit returns to a logical *0*. You "fire" the port by sending any value, including a *NULL* value (*0*) to the game port at address *201H*. You cannot fire an individual bit or one shot. When any value is sent to the port, all four one shots or lower bits will fire.

To check the *X* axis of joystick A, you can fire the one shots. In C, this is performed as follows:

```
outportb(0x201,0)
```

You must then check bit *0* of the port, which is the *X* axis of joystick A, for the returned value and ascertain when the bit has returned to a logical *0*. This is done with:

```
inportb((0x201) & 0x001))
```

```
Pins 1, 8, 9 & 15  +5 volts
Pins 4, 5, & 12  ground
Pin 3    joystick A, X axis (Potentiometer wiper)
Pin 6    joystick A, Y axis         "        "
Pin 11 joystcik B, X axis  "        "
Pin 13 joystick B, Y axis  "        "
Pin 2    joystick A, button A
Pin 7    joystick A, button B
Pin 10 joystick B, button A
pin 14 joystick B, button B
```

Figure 16.2 *Game port pinouts.*

The following code snippet shows the complete timing loop:

```
while (zap != (inportb(0x201) & 0x001))
/* zap is an arbitrary variable name */

      {
      ret++;
      }
return(ret);
```

To check the *Y* axis of joystick A, follow the same procedure, but check bit *2* of the returned value, which is:

```
(inportb(0x201) & 0x002))
```

To check the *X* axis of joystick B, check the third bit of the value returned:

```
(inportb(0x201) & 0x004))
```

To check the *Y* axis of joystick B, check the fourth bit of the return value. This would be:

```
(inportb(0x201) & 0x008))
```

Listing 16.1 shows a complete function which returns the value of the *X* axis on joystick A.

The R/C network fires at a constant rate and, for a given joystick setting, it takes the same amount of actual time on all machines for the capacitor to charge and for the port to normalize to a logical *0*. The actual return value, however, is a function of the processor clock speed and will vary from one machine to the next. On an 8088 XT class machine, the return values will be low. On an 80486, the return values can be quite high because the loop is executing faster. Compiler performance and efficiency can also change these values.

Using `clock()`

A more accurate way to time these joystick events might be to use the function *`clock()`* which is an ANSI C function available on all ANSI compatible compilers. The *`clock()`* function returns the actual time required for the port to normalize to logical *0*. In order to use the *`clock()`* function, you must include the header file *`time.h`*, set up a variable to accept the number of clock ticks, and set that variable to *0*. Then fire the one shot and start a loop. When the loop exits you can return the actual time it took for the loop to complete using the *`clock()`* function.

Using *`clock()`*, you can get a uniform timing for the value of the joystick that isn't dependent on the speed of the computer or the efficiency of the compiler. The *`clock()`* function will return the number of clock ticks that the procedure used. In order to use the *`clock()`* function you must declare variables that can accept the time value returned by *`clock()`*:

```
clock_t x,y,zap;
```

If you need to find out how many actual seconds have elapsed, you can divide our number of clock ticks by the value defined in the macro *`CLK_TCK`*, which is also defined in *`time.h`*.

Listing 16.2 shows a complete function using *`clock()`* that returns the *X*-axis of joystick B.

Returning Button Presses

The buttons on the joysticks are easier to check than the potentiometers because you do not need a timing sequence to get an approximate value. You need only check the proper bit and see whether it is high or low. In order to check button 1 of joystick A, you can again use the *`inportb()`* function as follows:

```
return = (0x010 != (inportb(0x201) & 0x010));
/* check the state of button 1 for joystick A */
```

To get the button values of the other joysticks, change the function call as follows:

- For button 2 of joystick A:
  `return = (0x020 != (inportb(0x201) & 0x020));`
- For button 1 of joystick B:
  `return = (0x040 != (inportb(0x201) & 0x040));`
- For button 2 of joystick B:
  `return = (0x080 != (inportb(0x201) & 0x080));`

Anytime a button is pressed, the function will return a *1*, (also known as *true*).

Microsoft C Compiler Users

For users of Microsoft compilers there is a change in the way the game port must be checked. Instead of using *inportb()* and *outportb()*, which is used in Borland and other compilers and is defined in *dos.h*, you must use *outp()* and *inp()*, which are defined in *conio.h* of Microsoft and compatible compilers.

An example in Microsoft C of getting the button value of button 1 of joystick A is as follows:

```
return = (0x010 != (inp(0x201) & 0x010));
```

An example in Microsoft C of getting a value of the *X*-axis of joystick A is:

```
outp(0x201,0);   /* Fire the one shots */
inp(0x201) & 0x001)  /* Retrieve the bit status */
```

Remember to include *conio.h* in your Microsoft C source files and to substitute *outp()* for *outportb()*. Substitute *inp()* for *inportb()*.

The Joystick Functions

In all of the libraries I make, I write each separate function in C and give each a function name and file name. I then prototype them in an `include` file for that library, compile the individual source modules for each function name, and make a library of functions that I can link in at compile time. The real world example, `joytest.c` in Listing 16.3, includes each joystick function in one source file.

The joystick functions used in Listing 16.3 will work for all types of computers, from an 8088 XT class computer through a 486 AT class computer. You may never know what kind of a computer your program will run on unless you compile it for a specific processor.

Anytime you use joystick routines, you should ask the user to push the joystick all the way down and to the right to obtain the highest values possible for the computer and then push the joystick all the way to the top and left so that the smallest values can be returned. As an example, let's say you are writing a program that runs in EGA 640 x 480 graphics mode. Your user's joystick, when pushed all the way down and to the right, returns a high *X*-axis value of 1280 and a high *Y*-axis value of 1440. From this point on, by dividing any joystick *X*-axis return value by 2, you have a very close screen pixel coordinate for the X-axis. By dividing any joystick *Y*-value by 3, you will then have a very close screen pixel coordinate for the *Y* axis. The joystick *X* and *Y* return values used in this example were selected for easy math, but these numbers can be realistic. Each user's computer is different both in CPU speed and port electrical

Register	**Contents**
AX	X-axis of joystick A
BX	Y-axis of joystick A
CX	X-axis of joystick B
DX	Y-axis of joystick B
AL	Button status in upper four bits of byte returned

Table 16.2 ***Function*** `84H` ***of AT compatable BIOS for joystick (or game port) values.***

characteristics and the results can vary greatly! This is why joystick calibration is a critical routine in any application that uses joystick input.

The AT BIOS Game Port Routines

When IBM came out with the AT they included BIOS support for the joystick port. This has carried through with some of today's BIOS ROMS, but not every modern BIOS manufacturer makes room for game port routines. In order to use the AT BIOS game port routines, you must check on function `84H` of `INT 15H`. Table 16.2 shows the registers that, on return of the interrupt function, hold the coordinates of the joystick.

In order to access the joystick coordinates you must first place a `1` in register `DX`. In C, you could use the `int86()` function prototyped in `DOS.H` to accomplish this task, but why? You already have a library of functions that will run on any class or type of MS-DOS computer and you don't have to rely on an AT compatible BIOS. Using the BIOS routines may, however, result in quicker execution of the function. An example using `int86()` is shown in Listing 16.4.

If you place a `0` in the `DX` register, you can ascertain the button status of the joysticks. The upper four bits, which are bits `4` through `7`, of the byte held in the `AL` register hold the value of the button presses. In order to find the button status you must check the specific bits for each button on each joystick as I did above in the low-level C routines by "`AND`ing" the value in `AL` to the bit you want to check. If the bit is *true*, the statement will be true and a `1` will be returned as in the following example:

```
ret = (0x010 != (btn & 0x010))
```

This will return the status of button 1 of joystick A.

Conclusion

Programming for the game port is not difficult. The game port can be very useful for things other than joysticks. The electronics experimenter or the computer control technician can easily use the game port for sensing real world events through digital input. The only thing the game port does not provide is output.

Control applications for the physically impaired must rely on the game port for some input and control. With some creativity and imagination, the game port can be used for a multitude of devices. Recently I have seen the button inputs of the game port used to calculate wind speed by sensing the revolutions of a wind cup. You can add a secondary numeric keypad to your computer by utilizing the button inputs of the game port. Frequency or pulse counting can easily be done through the game port. Positional sensing of a physical object can be done using the game port potentiometer inputs. A joystick can emulate mouse cursor movement, and the joystick will forever be with us in the games our children play!

Listing 16.1 *An example game port function.*

```
#include <dos.h>
static int zap;     /* static integer for value checking */
int getjoyAX(void)  /* Function Declaration */
{
int ret;            /* integer return value /*
ret = 0;
outportb(0x201,0)   /* Fire the one shot /*
while (zap != (inportb(0x201) & 0x001));
                    /* check the port and proper bit /*
{
    ret++;          /* loop /*
}
return(ret);         /* return the value or number of loops /*

/* End of File */
```

Listing 16.2 *Using* `clock()`.

```
#include <dos.h>
#include <time.h>

static int zap:
int getjoyBX(void)      /* Function name, declaration */
{
clock_t start, stop;    /* Declare start & stop as a time variable */
int i;
start = clock();        /* Get the start of the loop */
outportb(0x201,0);      /* Fire the one shot */
while (zap != (inportb(0x201) & 0x004));
                        /* Check the port for return to */
{
   i++;                 /* Do while loop */
}
stop = clock();         /* Get the number of ticks at the end of the loop */
return(stop - start);   /* Return the number of ticks */
}

/* End of File */
```

Listing 16.3 *Complete game port example program.*

```
/*-------------------------------------------------------------------------
        File: joytest.c
        Author: John F. Jarrett
        Copyright 1991, Computer Engineering Services
        This file was written for MIX Power C
        For Borland C, use goxy() to position the cursor instead of the
        MIX function poscurs()
-------------------------------------------------------------------------*/

#include <dos.h>
#include <bios.h>
#include <conio.h>

int x, y, b1, b2, xx, yy, b3, b4;     /* Assorted variables used */

/*-------------------------------------------------------------------------
        GETJOYAB1()
        This function returns a 1 or true, when the first button on
        the primary joystick is depressed. Otherwise it returns a 0,
        or false, when the function is called.
-------------------------------------------------------------------------*/
int getjoyAB1(void)
{
int ret;
 outportb(0x201,0);
 ret = (0x010 != (inportb(0x201) & 0x010));
 return(ret);
}

/*-------------------------------------------------------------------------
        GETJOYAB2()
        This function returns a 1 or true, when the second button on
        the primary joystick is depressed. Otherwise it returns a 0,
        or false, when the function is called.
-------------------------------------------------------------------------*/
int getjoyAB2()
{
int ret;
 outportb(0x201,0);
 ret = (0x020 != (inportb(0x201) & 0x020));
 return(ret);
}
```

Listing 16.3 *continued*

```
/*-----------------------------------------------------------------------
        GETJOYBB1()
        This function is identical to getjoyAB1 above, except returns a
        1, or true, if the first button on the secondary joystick
        is depressed.
-----------------------------------------------------------------------*/
int getjoyBB1(void)
{
int ret;
 outportb(0x201,0);
 ret = (0x040 != (inportb(0x201) & 0x040));
 return(ret);
}

/*-----------------------------------------------------------------------
        GETJOYBB2()
        This function is identical to getjoyAB2 above, except returns a
        1, or true, if the second button on the secondary joystick
        is depressed.
-----------------------------------------------------------------------*/
int getjoyBB2(void)
{
int ret;
 outportb(0x201,0);
 ret = (0x080 != (inportb(0x201) & 0x080));
 return(ret);
}

/*-----------------------------------------------------------------------
        GETJOYAX()
        This function returns the relative position of the primary
        joystick, (if you are using only one, this is the primary
        joystick), along its X axis. Returns a lower integer value
        when moved to the right and a higher value when moved to the
        left.
-----------------------------------------------------------------------*/
int getjoyAX(void)
{
int zap, ret;
 ret = 0;
 outportb(0x201,0);
 while (zap != (inportb(0x201) & 0x001))
 {
  ret++;
 }
 return(ret);
}
```

Listing 16.3 *continued*

```
/*-----------------------------------------------------------------------
       GETJOYAY()
       This function returns the relative position of the primary
       joystick along its Y axis. Returns a lower integer value when
       moved to the top and a higher value when moved to the bottom.
-----------------------------------------------------------------------*/
int getjoyAY(void)
{
int zap, ret;
 ret = 0;
 outportb(0x201,0);
 while (zap != (inportb(0x201) & 0x002))
 {
  ret++;
 }
 return(ret);
}

/*-----------------------------------------------------------------------
       GETJOYBX()
       This function is identical to getjoyAX above, except returns
       the X value of the secondary joystick
-----------------------------------------------------------------------*/
int getjoyBX(void)
{
int zap, ret;
 ret = 0;
 outportb(0x201,0);
 while (zap != (inportb(0x201) & 0x004))
 {
  ret++;
 }
 return(ret);
}

/*-----------------------------------------------------------------------
       GETJOYBY()
       This function is identical to getjoyAY above, except returns
       the Y value of the secondary joystick.
-----------------------------------------------------------------------*/
int getjoyBY(void)
{
int zap, ret;
 ret = 0;
 outportb(0x201,0);
 while (zap != (inportb(0x201) & 0x008))
 {
  ret++;
 }
 return(ret);
}
```

Listing 16.3 *continued*

```
main()
{
    clrscrn();
    poscurs(19,20);
       cprintf("PC Joy Stick Test, Copyright 1989 C.E.S.");
    poscurs(20,28);
       cprintf("Press any key to exit.");

 for(;;)
  {
  x = getjoyAX();
  y = getjoyAY();
  b1 = getjoyAB1();
  b2 = getjoyAB2();
  xx = getjoyBX();
  yy = getjoyBY();
  b3 = getjoyBB1();
  b4 = getjoyBB2();

         poscurs(5,10);
         cprintf("Joystick #1. X= %3d", x);
         poscurs(6,10);
         cprintf("Joystick #1. Y= %3d", y);
         poscurs(8,10);
         cprintf("Joystick #1, Button #1= %1d", b1);
         poscurs(9,10);
         cprintf("Joystick #1, Button #2= %1d", b2);
         poscurs(11,10);
         cprintf("Joystick #2. X= %3d", xx);
         poscurs(12,10);
         cprintf("Joystick #2. Y= %3d", yy);
         poscurs(14,10);
         cprintf("Joystick #2, Button #1= %1d", b3);
         poscurs(15,10);
         cprintf("Joystick #2, Button #2= %1d", b4);

   if (kbhit()) break;
   }
     clrscrn();
 }

 /* End of File */
```

Listing 16.4 *Using the AT BIOS.*

```
#include <dos.h>

static union REGS regs;
static void joystick(mode, jr1, jr2, jr3, jr4)
                                /*  function name  & prototype */
unsigned int mode, j2, j3, j4, *jr1, *jr2, *jr3, *jr4;

{
        regs.x.ah = 0x84;       /*  DOS function number */
        regs.x.dx = mode;       /*Place a 1 in DX to retrieve joystick
                                  position, 0 for btns */
        int86(0x15, &regs, &regs);     /* Call interupt 15h  */
        *jr1 = regs.x.ax;       /*  value of X axis of joystick A */
        *jr2 = regs.x.bx        /*  value of Y axis of joystick A  */
        *jr3 = regs.x.cx;       /*  value of X axis of joystick B  */
        *jr4 = regs.x.dx;       /*  value of Y axis of joystick B  */
        *btn = regs.x.al;       /*  value of the AL register for
                                    button presses  */
}

/* End of File */
```

Chapter 17

Using the Parallel Port for High-Speed File Transfer

David McCombs

This article will explore an uncommon use for one of the PC ports, the parallel port. The PC supports up to three parallel ports. Parallel ports were designed specifically to communicate with printers. This chapter will show how the parallel port can also be used as a PC-to-PC file transfer medium. The chapter will also suggest other uses. Utilizing the port as a high speed file transfer medium is the most likely and easiest alternative to implement. This use takes full advantage of the parallel port's strength: bandwidth. Bandwidth describes the amount of information passed per unit of time. A parallel port has the capability of transmitting data eight bits at a time, while the serial port can only transmit one. The parallel port has eight pins which were intended for data transmission. These leads use TTL logic, which makes them of low output impedance, which insures the leads have good immunity to noise and should be capable of high individual data speeds. The serial port has but one transmit and one receive pin. This means the serial port's bandwidth is limited to the maximum speed data bits can change at the transmit pin, travel the serial cable and then be read by the receiver without error. Due to the electrical nature of the RS-232 serial interface standard with its unbalanced and high impedance leads, the serial port's speed is limited.

Since the parallel interface has potential for high-speed communication and should be well suited for uses such as PC-to-PC file transfer, this chapter will explore the potential of the parallel port as a high-speed file transfer medium.

Objectives

My grandfather used to tell me, "Never use a tool for anything other than its intended purpose." In the context of his instruction, my grandfather was talking about farm tools, but the same caution applies to anything intended for a specialized use. If I am going to use the parallel port for something other than its intended purpose, I should start by formulating some clear objectives. For the purpose of this chapter, the unstated objective is instruction. The accompanying source code was written more for clarity than for optimization, though the software has nonetheless achieved good performance. The parallel port electrical interface will be described in detail, since it is the key to enabling 8-bit parallel data transmission. Here are the stated objectives for the copy program:

- ❍ Transmit a full eight bits of parallel data. Don't waste the parallel port's bandwidth by using less than eight bits. The objective is for data to flow by byte stream instead of bit stream.
- ❍ Design a parallel port symmetric null cable suitable for sending and receiving bytes of data.
- ❍ Design a protocol which will allow the sender and receiver to communicate indications of data ready and receive acknowledgement. This protocol will cause both ends to poll for "far" conditions to be met so that the next "near" action may be taken. No waits will be used in the software. This will allow faster machines to transmit files faster.

- Write all of the software in C (rather than assembly) with a minimal loss of execution speed.

A limitation: It won't be possible to send and receive eight bits of data simultaneously from both ends. The computer sending data will not be able to receive data until its sending is complete. This type of port capability is *half-duplex.*

Port Description

The parallel port is physically located on the back of an IBM PC, as a female 25-pin D shell connector. On older IBM computers, the port may be located on a monochrome display adapter. On a modern PC, it is

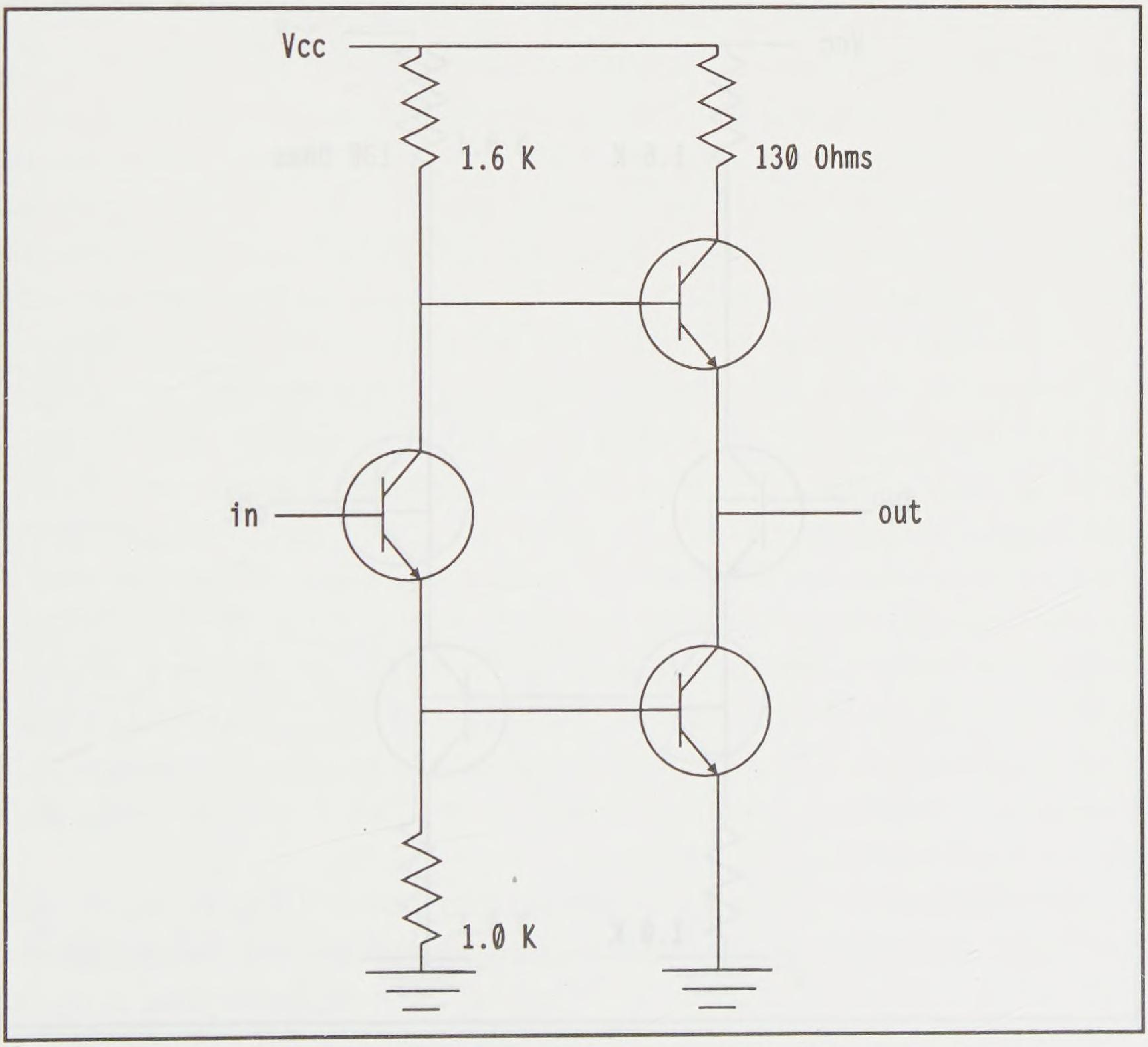


Figure 17.1 *Example TTL output circuit.*

located on an I/O card or on the back panel of the PC. The parallel port has four types of electrical connections on its 25 pins.

There are eight data outputs on pins 2 through 9. These pins are essentially write-only, with TTL type electrical output (Figure 17.1). These pins are powered by a 74LS374 chip. The 74 prefix means the chip is of the TTL family. The LS is for Low Power Schottky, which is a low power version of the TTL family.

Four control pins are read/write capable and have open collector electrical output provided by a 74LS174 chip (Figure 17.3). The parallel port card supplies resistors to each output to make the open collector circuit look like Figure 17.2. The inputs are fed to a 74LS240 chip, which reads TTL logic levels. These pins have the following functions:

- Pin 1, the STROBE pin, paces the flow of the data.
- Pin 14, the AUTOFEED pin, sets the printer to automatically form feed.

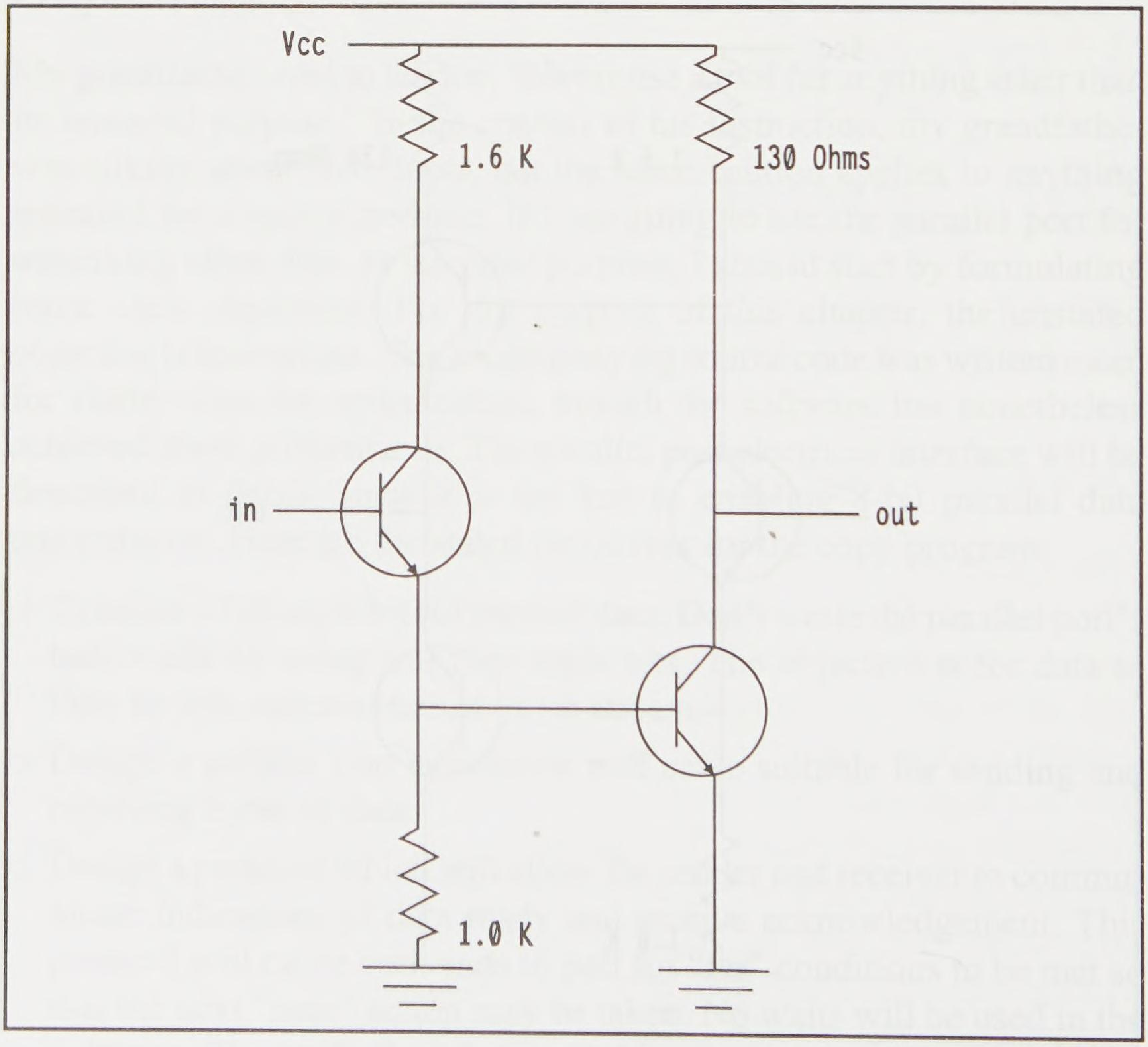


Figure 17.2 *Example OCR output circuit.*

- Pin 16, the INIT pin, initializes the printer.
- Pin 17 the SLCT IN pin, conditions the printer to accept data.

Five status pins are read-only. The inputs are read by a 74LS240 chip. These pins read the following conditions at TTL logic levels:

- Pin 15, the ERROR pin, reads whether a printer error has occurred.
- Pin 13, the SLCT pin, reads whether the printer is in the selected state.
- Pin 12 is the PE pin. It reads whether the printer is out of paper.
- Pin 10 is the ACKNLG pin. It indicates that the printer has received the data and is ready to receive more.
- Pin 11, the BUSY pin, indicates that the printer cannot receive data.

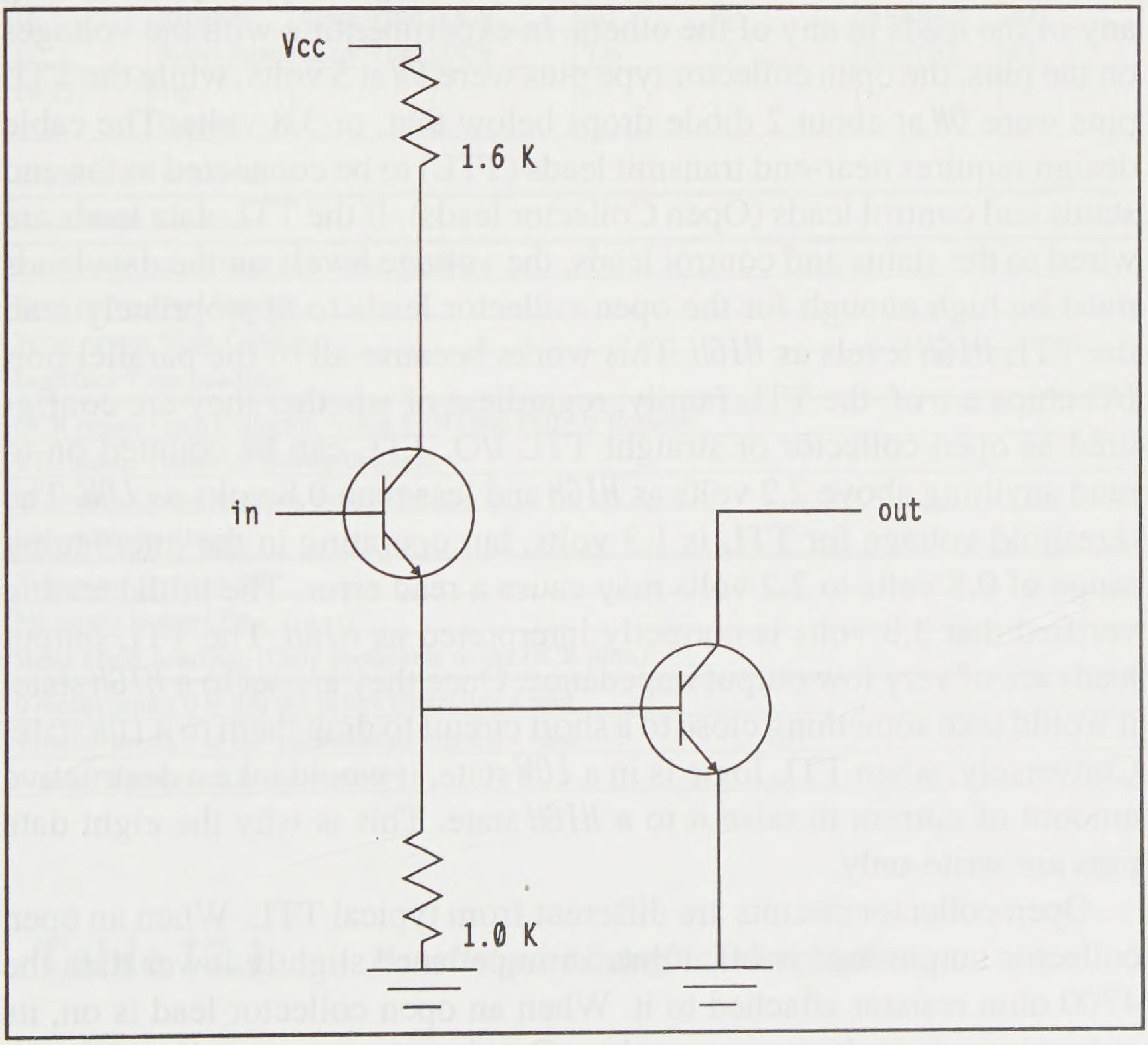


Figure 17.3 ***Example OC output circuit.***

Pins 18 through 25 are signal ground pins. These pins were intended to be used as the ground component in the transmit pins (2-9) twisted wire pairs. In cables, twisted pairs are used to cancel noise caused by magnetic induction. There are 17 active pins and only eight ground pins. The design for the synchronous null cable (presented later in this chapter) used in the parallel file transfer uses these ground pins more than once in order to have them serve as the ground component for all 17 twisted wire pairs needed for half-duplex communication.

Electrical Characteristics

Any of the active pins will tolerate voltage levels between 0 and 5.2 volts regardless of their output state. The maximum output on any of the leads is 5.0 volts. This means it is not possible to damage the port by connecting any of the leads to any of the others. In experimenting with the voltages on the pins, the open collector type pins were `ON` at 5 volts, while the TTL pins were `ON` at about 2 diode drops below that, or 3.8 volts. The cable design requires near-end transmit leads (TTL) to be connected to far-end status and control leads (Open Collector leads). If the TTL data leads are wired to the status and control leads, the voltage levels on the data leads must be high enough for the open collector leads to appropriately read the TTL `HIGH` levels as `HIGH`. This works because all of the parallel port I/O chips are of the TTL family, regardless of whether they are configured as open collector or straight TTL I/O. TTL can be counted on to read anything above 2.2 volts as `HIGH` and less than 0.8 volts as `LOW`. The threshold voltage for TTL is 1.3 volts, but operating in the intervening range of 0.8 volts to 2.2 volts may cause a read error. The initial testing verified that 3.8 volts is correctly interpreted as `HIGH`. The TTL output leads are of very low output impedance. Once they are set to a `HIGH` state, it would take something close to a short circuit to drag them to a `LOW` state. Conversely, when TTL logic is in a `LOW` state, it would take a destructive amount of current to raise it to a `HIGH` state. This is why the eight data pins are write-only.

Open collector circuits are different from typical TTL. When an open collector output lead is off, it has an impedance slightly lower than the 4700 ohm resistor attached to it. When an open collector lead is on, its output impedance becomes very low. Data buses use open collector logic. This makes the four read/write control pins on the port very much a 4-bit

Pin Number	**Traditional Use**	**Read/ Write**	**Interface Type**	**Invert Pin**	**Read Mask**	**Por Address**	**Port Bit**
1	STROBE	Read/Write	OCR	Yes	0	BASE+2	B0
2	D0	Write	TTL	No	-	BASE	B0
3	D1	Write	TTL	No	-	BASE	B1
4	D2	Write	TTL	No	-	BASE	B2
5	D3	Write	TTL	No	-	BASE	B3
6	D4	Write	TTL	No	-	BASE	B4
7	D5	Write	TTL	No	-	BASE	B5
8	D6	Write	TTL	No	-	BASE	B6
9	D7	Write	TTL	No	-	BASE	B7
10	ACK	Read	OC	No	-	BASE+1	B6
11	BUSY	Read	OC	Yes	-	BASE+1	B7
12	Paper End	Read	OC	No	-	BASE+1	B5
13	Select	Read	OC	No	-	BASE+1	B4
14	Auto Feed	Read/Write	OCR	Yes	0	BASE+2	B1
15	Error	Read	OC	No	-	BASE+1	B3
16	Initialize	Read/Write	OCR	No	1	BASE+2	B2
17	Select Input	Read/Write	OCR	Yes	0	BASE+2	B3
18-25	Sig. Ground						

Explanation of Terms

Read/Write heading:

Read/Write means Read and Write Capable.

Read means Read Capable.

Write means Write Capable.

Interface Type heading:

OCR means Open Collector, with a 4700 Ohm Pull-Up Resistor.

TTL means Transistor Transistor Logic.

OC means Open Collector, Read Only.

Invert Pin heading:

Yes means Inverted Pin.

No means Normal Pin.

Read Mask heading: (Only applicable to the OCR pins.)

0 means send a 0 to this pin before attempting a read.

1 means send a 1 to this pin before attempting a read.

- means a read mask is not necessary.

Table 17.1 ***Parallel port interface description.***

data bus. This means these pins could be connected pin-for-pin across two separate machines and, providing the reading machine had its own pins in the correct state, either machine could read the other's output.

The five status pins are read-only and expect TTL input levels. Another aspect of the port is that some of the pins, pins 1, 11, 14, and 17, are inverted. This means a positive voltage on a pin is interpreted as a logical `0`, while no voltage is interpreted as a logical *1*. Three of the inverted pins, 1, 14, and 17, are read/writable control pins. The remaining pin, 11, is a read-only status pin. The read/writable "bus" type pins need to be asserted from the receiving end with the correct logic level in order to read data from the transmitter. The reason for this is that the pin must float its open collector to keep the read voltage from being pulled down. Otherwise, the effect would be that the read data is destructively `OR`ed with the pin's output voltage. For inverted bus-style pins, the correct logic level for floating the open collector is `0`. For non-inverting pins the correct logic level is *1*. These logic levels are set in the program by sending a "read mask" byte to the port address with the open collector pins. Table 17.1 provides a complete description of the port's pins and electrical interfaces.

Port Access

Port access may be achieved by communicating with the port by port addresses and address offsets. Accessing the port base address sets the write pins (2 through 9). The status pins are read at the base address plus an offset of one byte. The control pins are set or read at the base address plus an offset of two bytes.

The base address for the typical PC `LPT1` port is `Hex 378`. The base address for the typical PC `LPT2` port is `Hex 278`. The base address for the IBM XT Monochrome display adapter parallel port is `Hex 3BC`.

Several functions provide general port I/O. The most versatile are the Microsoft `inpw(address)`, `inp(address)`, `outpw(address, word)`, `outp(address, byte)`, and the Borland `inportb(address)` `inport(address)`, `outportb(address,byte)`, and `outport(address,word)` functions. These functions are similar to the assembly instructions for port I/O, `in` and `out`.

Eight Bits Half-duplex

This file transfer program is limited to eight bits half-duplex. Given the configuration of the parallel port adapter, the port can output on up to 12 pins, but only if the input is limited to five pins. Alternately, up to nine pins can be used for input, while using eight pins for output, but this latter configuration offers no possibility of controlling the flow of data. A solution is to use the eight data lines for the output of data, while using the strobe for (bi-directional) indication of data ready. That leaves any of the eight remaining active pins for the receiving computer to send back acknowledgment of data received and indications of ready to receive more data.

Sending and Receiving Bytes

In order to send or receive data through the parallel port, some semblance of order must be made of the interface. Since the data output leads ascend in order from pins 2 through 9, they won't pose a problem. Making sense of the pins to be used for reading, the status and control pins, does pose a problem. Which pins should be used for reading the data and in what order? After eliminating the output pins, only nine active pins are left. Attempting to make sense of the pins in their numeric pin order is a mistake. The pins' locations with respect to their functions seems ran-

Pin Number	Traditional Use	Read/ Write	Interface Type	Invert Pin	Read Mask	Port Address	Port Bit
15	Error	Read	OC	No	-	BASE+1	B3
13	Select	Read	OC	No	-	BASE+1	B4
12	Paper End	Read	OC	No	-	BASE+1	B5
10	ACK	Read	OC	No	-	BASE+1	B6
11	BUSY	Read	OC	Yes	-	BASE+1	B7
1	STROBE	Read/Write	OCR	Yes	0	BASE+2	B0
14	Auto Feed	Read/Write	OCR	Yes	0	BASE+2	B1
16	Initialize	Read/Write	OCR	No	1	BASE+2	B2
17	Select Input	Read/Write	OCR	Yes	0	BASE+2	B3

Table 17.2 ***The read pins arranged by address and bit orders.***

dom. (This may have been done in an effort to limit crosstalk.) It is more fruitful to look at pins as they are logically addressed and by their bit position in the returned bytes. The pin numbering disarray is left where it belongs, as a parallel cable wiring problem. Rearranging the read-capable pins of Table 17.1 by port address and bit number yields a second, more manageable table, Table 17.2.

Notice that on Table 17.2, the pins accessed from the `BASE + 1` address go from bits `3` to `7`. The pins accessed from the `BASE + 2` address go from bits `0` to `3`. Following the "use tools for their intended purpose" axiom, I decided to use the strobe as the strobe. When I use the strobe in its intended role as the data pacer, I am left with the `BASE + 1` addressed pins appearing from bits `3` to `7` and the `BASE + 2` addressed pins appearing from bits `1` to `3`. If I right-shift the bits from the `BASE + 2` address, they would appear from bits `0` to `2`. This makes these bits readily available to be bitwise `OR`ed with the upper portion of the byte from the `BASE + 1` address. Table 17.3 illustrates this implementation methodology.

Pin Number	Traditional Use	Read/ Write	Interface Type	Invert Pin	Read Mask	Port Address	Port Bit
(Use the strobe to indicate a data ready condition.)							
1	STROBE	Read/Write	OCR	Yes	0	BASE+2	B0
(Right shift the three remaining BASE+2 pins by one bit.)							
1	Auto Feed	Read/Write	OCR	Yes	0	BASE+2	B1>>B0
16	Initialize	Read/Write	OCR	No	1	BASE+2	B2>>B1
17	Select Input	Read/Write	OCR	Yes	0	BASE+2	B3>>B2
(Bit-wise OR the result with the five read only pins at BASE+1.)							
15	Error	Read	OC	No	-	BASE+1	B3
13	Select	Read	OC	No	-	BASE+1	B4
12	Paper End	Read	OC	No	-	BASE+1	B5
10	ACK	Read	OC	No	-	BASE+1	B6
11	BUSY	Read	OC	Yes	-	BASE+1	B7

Table 17.3 *The read pins as used by the file transfer software.*

The Protocol

The protocol must provide a means of sending the data with adequate synchronization between the sending and receiving computers. The data must reach its target without error regardless of any capability differences between the sending and receiving machines. This requires that the transmitter set an indication of data sent and the receiver set an acknowledgement of data received, as well as a ready status. For this scheme you need a minimum of two pins, one for the data ready, and one for dual use (acknowledgement and receiver ready). The file copy program uses the (bi-directional) STROBE pin for the data ready indication, and the BUSY pin for the acknowledge and receiver ready indications. The transmitter and receiver are locked in sequence with each other as shown in Figure 17.4.

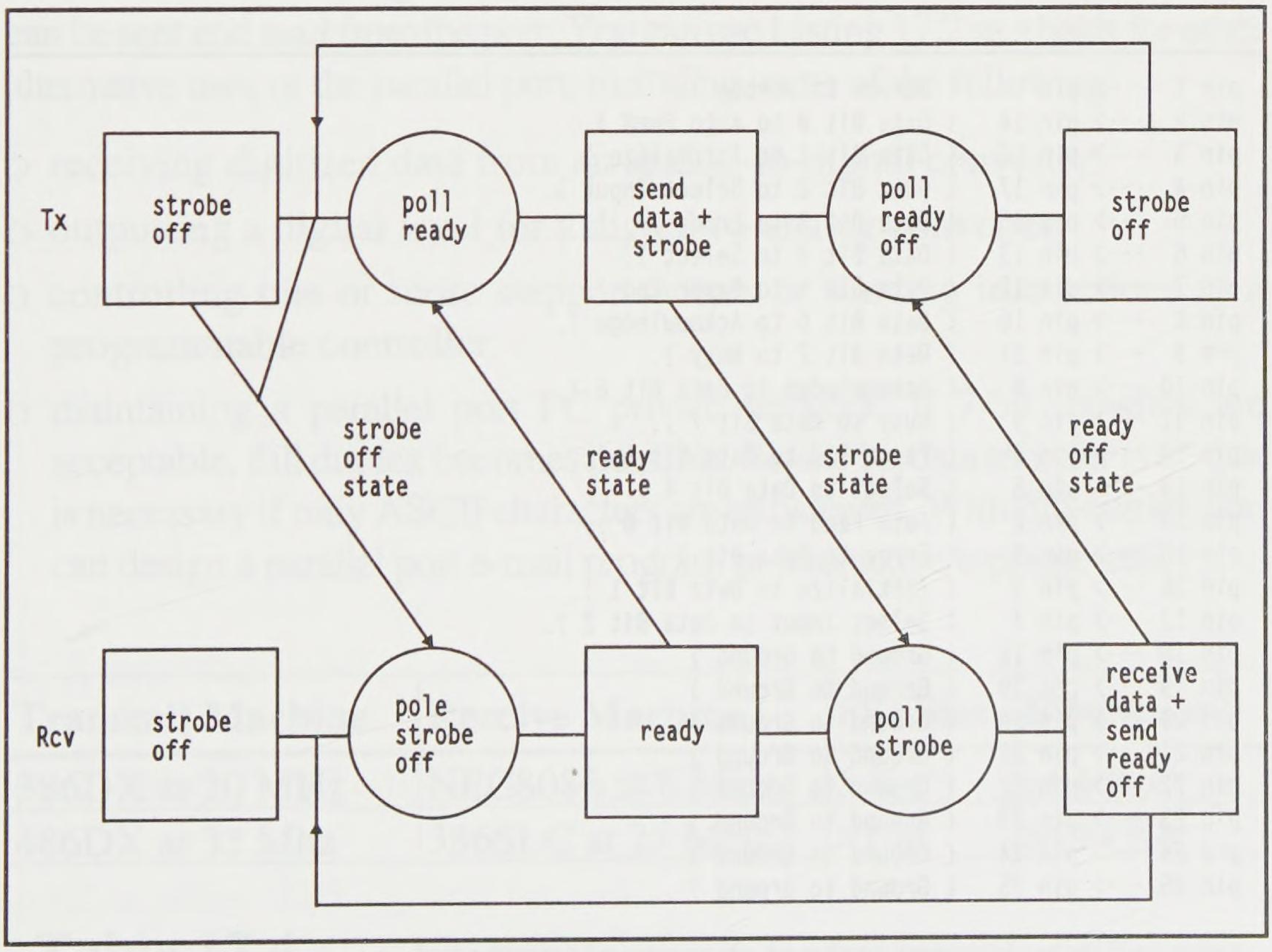

Tx
strobe off
poll ready
send data + strobe
poll ready off
strobe off
strobe off state
ready state
strobe state
ready off state
Rcv
strobe off
pole strobe off
ready
poll strobe
receive data + send ready off

Figure 17.4 ***Parallel port polling sequence.***

The Parallel Null Cable

The parallel null cable can be made with a cable having 25 twisted wire pairs and two DB-25 male connectors. Better results and longer cable lengths are possible if the cable has a ground shield. These parts can be found at many electronics and service supply stores. Some DB-25 connectors require a soldering iron. You only need 17 of the 25 pairs. In order to eliminate induced noise, one wire of each pair should be used as the active lead, and the other should be used as the ground lead. Be sure not to reverse the wires in a pair or the active lead will be grounded. Figure 17.5 details the pin-for-pin connections to the cable.

The File Transfer Program

paracopy.c is the source file for the file transfer program (see Listing 17.2). The header file, *paracopy.h*, is shown in Listing 17.3. The send routine

```
pin 1  ---> pin 1    ( Strobe to Strobe ).
pin 2  ---> pin 14   ( Data Bit 0 to Auto Feed ).
pin 3  ---> pin 16   ( Data Bit 1 to Initialize ).
pin 4  ---> pin 17   ( Data Bit 2 to Select Input ).
pin 5  ---> pin 15   ( Data Bit 3 to Error ).
pin 6  ---> pin 13   ( Data Bit 4 to Select ).
pin 7  ---> pin 12   ( Data Bit 5 to Paper End ).
pin 8  ---> pin 10   ( Data Bit 6 to Acknowledge ).
pin 9  ---> pin 11   ( Data Bit 7 to Busy ).
pin 10 ---> pin 8    ( Acknowledge to Data Bit 6 ).
pin 11 ---> pin 9    ( Busy to Data Bit 7 ).
pin 12 ---> pin 7    ( Paper End to Data Bit 5 ).
pin 13 ---> pin 6    ( Select to Data Bit 4 ).
pin 14 ---> pin 2    ( Auto Feed to Data Bit 0 ).
pin 15 ---> pin 5    ( Error to Data Bit 3 ).
pin 16 ---> pin 3    ( Initialize to Data Bit 1 ).
pin 17 ---> pin 4    ( Select Input to Data Bit 2 ).
pin 18 ---> pin 18   ( Ground to Ground )
pin 19 ---> pin 19   ( Ground to Ground )
pin 20 ---> pin 20   ( Ground to Ground )
pin 21 ---> pin 21   ( Ground to Ground )
pin 22 ---> pin 22   ( Ground to Ground )
pin 23 ---> pin 23   ( Ground to Ground )
pin 24 ---> pin 24   ( Ground to Ground )
pin 25 ---> pin 25   ( Ground to Ground )
```

Figure 17.5 *The parallel port cable wiring scheme.*

on the sending computer transfers data by filling a 30Kb byte buffer from the disk and then sending the buffer to the port one byte at a time. The receive routine active on the receiving computer reads the data from the port into a 30Kb byte buffer and writes the data to disk. The program uses a simple protocol to control the flow of the data. The program is bare bones with little error handling. The source machine will hang in a polling loop if the receiving machine does not send proper acknowledgment. If the source or target machine hangs, the only alternative is to reboot both machines. The program will not handle input of a filename on the source machine with a fully qualified path that does not exist on the target machine. Checks for these events can easily be added but are not within the scope of this chapter.

The Test Harness

Listing 17.1, `paraport.c`, is the source file for a test harness that tests the capability of sending and receiving a full byte from the parallel port using the BUSY and STROBE pins for controlling data flow. The program is simpler than the file transfer program and is intended to demonstrate how bytes of data can be sent and read from the port. You can use Listing 17.2 as a basis for other alternative uses of the parallel port, including some of the following:

- receiving digitized data from an analog-to-digital converter.
- outputting a digital level for a digital-to-analog converter.
- controlling one or more stepper motors or sending instructions to a programmable controller.
- maintaining a parallel port PC private network. If 7-bit transfers are acceptable, full duplex becomes possible. Seven-bit data transfer is all that is necessary if only ASCII characters are transferred. With full duplex you can design a parallel port e-mail program or interactive e-phone utility.

Transmit Machine	Receive Machine	File Size	Bits/Second
386DX at 20 MHz	NEC8086 at 8 MHz	5 K	>64 Kb
486DX at 33 Mhz	386SLC at 25 MHz	>1 M	>144 Kb

Table 17.4 ***File transfer speeds for two typical parallel port configurations.***

Conclusion

The parallel port was originally intended only for printers, but programmers have found it useful for much more. Table 17.3 shows the file transfer speeds for two typical configurations using a parallel port file transfer scheme.

The data was collected with an executable compiled using Borland's Turbo C compiler with no execution optimizing options.

Improvements

In this article I have not discussed the parallel port interrupt. The interrupt `IRQ7` is generated when the ACK pin (pin 7) is made low. I have read articles which use the interrupt for the flow of data, but in all such cases the author has indicated that the polling method is faster. A reasonable use of the interrupt would be in a Terminate-and-Stay-Resident (TSR) program which waits for the parallel port interrupt and becomes active when the port interrupt occurs. The monitor program could then check to see if a legitimate file transfer is taking place and begin receiving data or return to the monitor state in the case of a normal print operation.

One possible improvement to the file transfer program is to poll for pin state transitions instead of states. This could cut the number of port accesses in half. Another improvement would be to read the data and the strobe during the polling loop by reading the whole word at the `BASE + 1` address. This would eliminate yet another port access. Invoking performance optimization or writing port access polling loops in assembly code may further improve performance.

References

The Art of Electronics, Paul Horowitz, Winfield Hill, Cambridge University Press, 1980.

Eckel, Bruce. "A Programmers Guide to the Parallel Port," Turbo Technix, November/December, 1987.

Greenburg, Ross M. "Adapting the Parallel Port for Bidirectional Communications," Microsoft Systems Journal, September, 1990.

The IBM PC-XT Technical Reference. IBM Corporation.

Listing 17.1 paracopy.c

```
/*
   Paraport.c
   1993 by David McCombs
   for R&D Publications, MS-DOS System Programming, III Edition.
   This program is used as a test harness for sending and recieving
   characters through the Parallel Port.
*/

#include <stdio.h>

#include <dos.h>

#include <bios.h>

#include <ctype.h>

#include "paracopy.h"

unsigned char GetParaChar( int base )
{

 byte bLowByte = 0;

 byte bHiByte = 0;

 byte bCombinedByte = 0;

 byte bFinishedByte = 0;

 /*
    The transmitter should have its strobe low, wait for it.
 */
 while ( !( STROBE ( inportb( base + 2 ) ) ) );

 /*
    Output the read mask to the port base address + offset 2
 */
 outportb( ( base + 2 ), READMASK );

 /*
    Let the transmitter know we are ready.
 */
 outportb( base, 0x80 );
```

Listing 17.1 *continued*

```
/*
   poll for the strobe to go high. An inverted strobe pin means the pin value will be
   low when logical value is high. As little is done as is possible in the polling loop.
*/
do
{
 /*
    get the strobe and the low three bits of data
 */
 bLowByte = inportb( base +  2);
}
while ( STROBE( bLowByte ) );

/*
   Must have gotten a byte.
*/

/*
   First, shift the byte to the right one bit to get rid of the strobe bit.
*/
bLowByte = SHIFT( bLowByte );

/*
    Next, get the the other byte from offset 2
*/
bHiByte = inportb( base + 1 );

/*
   Next, combine the bytes.
*/
bCombinedByte = ( bLowByte & 0x07 ) | ( bHiByte & 0xF8 );

/*
   Next, Condition the byte by masking out and operating on inverted pins.
*/
bFinishedByte = ( NORMAL( bCombinedByte ) | INVERTED( bCombinedByte ) );

/*
   Make the transmitter wait and return.
*/
outportb( base, 0x00 );

return bFinishedByte;

}
```

Listing 17.1 *continued*

```
void PutParaChar( int base, unsigned char chPut )
{

 /*
    Wait for ready state.
 */
 while( ( inportb( base + 1 ) & 0x80 ) ) ;

 /*
    Ouput the data
 */
 outportb( base, chPut );

 /*
    Send strobe
 */
 outportb( base + 2, STROBELOW );

 /*
    Wait for not ready on char read as an acknowledgment.
 */
 while( ! ( inportb( base + 1 ) & 0x80 ) ) ;

 /*
    Turn the strobe off for the next operation.
 */
 outportb( base + 2, STROBEHIGH );

}

int main( void )
{

 unsigned char chIO;
 unsigned char chParaChar;

 outportb( BASE1 + 2, STROBEHIGH );

 system("cls");
```

Listing 17.1 *continued*

```
do
 {

  printf("\n(I)nput, (O)utput or (Q)uit?\n ");

  scanf("%c%*c",&chIO);
  fflush( stdin );
  chIO = toupper( chIO );

  if ( chIO == 'I' )
  {

   printf( "\nCharacters from parallel port:\n ");

   while ( (chParaChar = GetParaChar( BASE1 )) != '~' )
    putc( chParaChar, stdout );

  }
  else if ( chIO == 'O')
  {

   printf( "Enter characters to send to parallel port\n");

   printf( "Enter (~) to stop sending.\n");

   while ( 1 )
   {

    while ( bioskey( 1 ) );

    chParaChar = ( char ) bioskey(0);

    PutParaChar( BASE1, chParaChar );

    if ( chParaChar == '~' )
     break;

   } /* end of infinite loop */

  } /* end of if output */

 } while( chIO != 'Q' );

 return (0);

}

/* End of File */
```

Listing 17.2 paraport.c

```
/*
   1993 by David McCombs
   for R&D Publications, MS-DOS System Programming, III Edition.
   This program is used as a test harness for sending
   and recieving characters through the Parallel Port.
*/

/*
   To compile with a Microsoft compiler, un-comment the MSC_COMPILER definition.
*/

/*

#define MSC_COMPILER

*/

#include "paracopy.h"

#include <stdio.h>

#include <io.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <dos.h>

#include <time.h>

#include <bios.h>

#include <ctype.h>

#include <string.h>

/* function prototypes */

unsigned long GetFarFile( int base );
unsigned long PutNearFile( int base );

void PortToBuffer( int base, byte * buffer, unsigned long size, int IsFileInfo );
void BufferToPort( int base, byte * buffer, unsigned long size, int IsFileInfo );
```

Listing 17.2 *continued*

```
/*
   Function: GetFarFile( int base )

   Description: GetFarFile Gets the file size and name from the transmitting
   machine, opens a file of the same name and writes the transmitted
   data to the open file.  The file is closed when all the data is recieved.

   Returns: (unsigned long) file size.

*/
unsigned long GetFarFile( int base )
{

 byte * pbBufCursor = TransferBuffer;

 unsigned long * pulSize;

 int iFileHandle;

 int iBlocks, iBlockIndex, iRemainder;

 /*
    Get the file name and size
 */
 PortToBuffer( base, pbBufCursor, 300, 1 );

 /*
    Extract the file size from the first four bytes.
 */
 pulSize = ( unsigned long * ) pbBufCursor;

 FileSize = *pulSize;

 /*
    Extract the File Name;
 */
 pbBufCursor+=4;
 sprintf( FileName, pbBufCursor );

 printf(" File = %s, Size = %d\n", FileName, FileSize );

 /*
    Attempt to open the file for writing binary data.
 */
 iFileHandle = open( FileName, O_CREAT | O_WRONLY | O_BINARY, _IREAD | S_IWRITE );

 if ( iFileHandle == -1 )
 {
  printf( "File %s cannot be opened.\n", FileName );
  exit ( 1 );
 }
```

Listing 17.2 *continued*

```
/*
   Derive the number of blocks and the remainder.
 */

 iBlocks = FileSize / BUFFERSIZE;
 iRemainder = FileSize % BUFFERSIZE;

 printf( "Blocks: %d, Remainder : %d\n", iBlocks, iRemainder );

 for ( iBlockIndex = 0; iBlockIndex < iBlocks; iBlockIndex++ )
 {
  /*
    Get a block of data from the port and write it.
  */
  PortToBuffer( base, TransferBuffer, BUFFERSIZE, 0 );

  write( iFileHandle, TransferBuffer, BUFFERSIZE );

 }

 /*
    Finish with the remainder, if any.
 */
 if ( iRemainder )
 {
  PortToBuffer( base, TransferBuffer, iRemainder, 0 );

  write( iFileHandle, TransferBuffer, iRemainder );
 }

 close( iFileHandle );

 return FileSize;
}
```

Listing 17.2 *continued*

```
/*
   Function: PutNearFile( int base )

   Description: PutNearFile opens the file with the filename stored in global
   FileName string. It gets the files size, and writes it to the global byte
   buffer. The function then sends the buffer to the port with the file size
   and name so the recieving end can open a file. The function then sends the
   file data, closes the file, and returns.

   Returns: (unsigned long) file size.
*/

unsigned long PutNearFile( int base )
{

 /*
    Set the buffer pointer to the start of the (global) transfer buffer.
 */
 byte * pbBufCursor = TransferBuffer;

 /*
    Define a pointer to an unsigned long type so we can put the file size
    into the byte array buffer as a unsigned long entity.
 */
 unsigned long * pulSize;

 int iFileHandle;

 /*
    Define variables for block transfer of the data.
 */
 int iBlocks, iBlockIndex, iRemainder;

 /*
    Define variables for performance measure.
 */
 clock_t begin, end;

 float elapsed;

 /*
    Check for file existance.
 */
 if( access( FileName, 0 ) )
 {
  printf( "File %s not found", FileName );
  exit( 1 );
 }
```

Listing 17.2 *continued*

```
/*
   Open the file
*/

iFileHandle = open( FileName, O_RDONLY | O_BINARY );

if ( iFileHandle == -1 )
{
 printf( "File %s cannot be opened.\n", FileName );
 exit ( 1 );
}

/*
   Get the size of the file
*/

FileSize = filelength( iFileHandle );

printf(" File = %s, Size = %ld\n", FileName, FileSize );

/*
   Put the file size in the first four bytes of the buffer.
*/
pulSize = (unsigned long *) pbBufCursor;

*pulSize = FileSize;

/*
   Put the file name next with a null termination;
*/
pbBufCursor += 4;
sprintf( pbBufCursor, FileName );

/*
  Send the data to the port.
*/
BufferToPort( base, TransferBuffer, 300, 1 );

/*
   Calculate the number of data blocks and any remainder.
*/
iBlocks = FileSize / BUFFERSIZE;
iRemainder = FileSize % BUFFERSIZE;

printf( "Blocks: %d, Remainder : %d\n", iBlocks, iRemainder );
```

Listing 17.2 *continued*

```
 /*
    Get the start time.
 */
 begin = clock();

 for ( iBlockIndex = 0; iBlockIndex < iBlocks ; iBlockIndex++ )
 {
  /*
     Get a block of data from the file, then send it to the port.
  */
  read( iFileHandle, TransferBuffer, BUFFERSIZE );

  BufferToPort( base, TransferBuffer, BUFFERSIZE, 0 );
 }

/*
    Finish with the remainder, if any.
 */
 if ( iRemainder )
 {

  read( iFileHandle, TransferBuffer, iRemainder );

  BufferToPort( base, TransferBuffer, iRemainder, 0 );

 }
 /*
    Get the stop time
 */
 end = clock();

 close( iFileHandle );

 /*
    Calculate and report the time taken to read, and send the file.
 */

 elapsed = ( (float) ( end - begin ) ) / CLK_TCK;

 if ( elapsed > 0.0 )
 {
  printf("Transmission time: %g\n", elapsed );
  printf("Bytes per Second: %g\n", ((float) FileSize) / elapsed );
 }

 return FileSize;
}
```

Listing 17.2 *continued*

```
/*
   Function: PortToBuffer( int base, byte * buffer, unsigned long size, int IsFileInfo )

   Description: PortToBuffer fills the passed 'buffer' with bytes from prot 'base.'
   It returns after 'size' bytes have been passed, unless 'IsFileInfo' is true,
   in which case it will return as soon as the first 0 after the fourth byte is
   encountered. The first four bytes hold the file size.

   Returns: (void).
 */

void PortToBuffer( int base, byte * buffer, unsigned long size, int IsFileInfo )
{

 long int iIndex;

 /*
    Condition the OCR pins once.
 */
 outportb( ( base + 2 ), READMASK );

 for ( iIndex = 0; iIndex < size; iIndex++ )
 {

  /*
    Transmitter strobe should be low until a ready is sent from here.
    Make sure it has transitioned to low.
  */
  while ( !( STROBE ( inportb( base + 2 ) ) ) );

  /*
    Send Ready signal
  */
  outportb( base, 0x80 );

  /*
    poll for the strobe to go high.
  */
  do
  {
   bLowByte = inportb( base +  2);
  }
  while ( STROBE( bLowByte ) );
```

Listing 17.2 *continued*

```
   /*
      shift the byte to the right one bit to get rid of the strobe bit.
   */
   bLowByte = SHIFT( bLowByte );

   /*
      Get the the other byte from offset 2
   */
   bHiByte = inportb( base + 1 );

   /*
      Combine the bytes.
   */
   bCombinedByte = ( bLowByte & 0x07 ) | ( bHiByte & 0xF8 );

   /*
      Condition the byte by masking out and operating on inverted pins.
   */
   bFinishedByte = ( NORMAL( bCombinedByte ) | INVERTED( bCombinedByte ) );

   /*
      Make the transmitter wait until we loop around. Ready is made low.
   */
   outportb( base, 0x00 );

   *buffer++ = bFinishedByte;

   if ( IsFileInfo && ( bFinishedByte == (byte) 0 ) && iIndex >= 4 )
    break;

  }

}
```

Listing 17.2 *continued*

```
/*
   Function: BufferToPort( int base, byte * buffer, unsigned long size, int IsFileInfo )

   Description: BufferToPort takes bytes from the 'buffer' and sends them to the port
   at 'base'. It returns after 'size' bytes have been passed or after seeing a 0 in the
   buffer somewhere after the fourth byte in the buffer and when 'IsFileInfo' is true.

   Returns: (void).
*/

void BufferToPort( int base, byte * buffer, unsigned long size, int IsFileInfo )
{
 int iIndex;

 for ( iIndex = 0; iIndex < size; iIndex++ )
 {
  /*
     Poll for ready from reciever.
  */
  while( ( inportb( base + 1 ) & 0x80 ) );

  /*
     Ouput the data
  */
  outportb( base, *buffer++ );

  /*
     Send the strobe
  */
  outportb( base + 2, STROBELOW );

  /*
     poll for not ready as acknowledgment.
  */
  while( ! ( inportb( base + 1 ) & 0x80 ) ) ;

  /*
     Turn the strobe off for the next operation.
  */
  outportb( base + 2, STROBEHIGH );

  if ( IsFileInfo && ( *(buffer-1) == (byte) 0 ) && ( iIndex >= 4 )  )
   break;
 }

}
```

Listing 17.2 *continued*

```
/*
   Function: main( int iArgCount, char *pszArgVerb[] )

   Description: Insures 2 to 3 arguments are passed, the second argument,
   'pszArgVerb[1]' is checked to see if input or outut is requested. If
   input is requested main calls GetFarFile. If output is requested, main
   checks for a file name, and if one exists, calls PutNearFile.

   Returns: (int) exit status.
*/

int main( int iArgCount, char *pszArgVerb[] )
{

 if ( iArgCount < 2 || iArgCount > 3 )
 {
  printf("USAGE: Paraport (I/O) <File Name>\n");

  exit( 1 );

 }

 outportb( BASE1 + 2, STROBEHIGH );

 if ( !strcmp(pszArgVerb[1], "I") || !strcmp(pszArgVerb[1], "i") )
 {
  GetFarFile( BASE1 );
 }

else if ( !strcmp(pszArgVerb[1], "O") || !strcmp(pszArgVerb[1], "o") )
 {
   if( iArgCount != 3 )

   {
    printf("USAGE: Paraport (I/O) <File Name>\n");

    exit( 1 );
   }
   sprintf( FileName, pszArgVerb[2] );
   PutNearFile( BASE1 );
 }

 return (0);

}

/* End of File */
```

Listing 17.3 paracopy.h

```
/*

 Paracopy.h
 Definitions for paracopy.c

*/

#define BUFFERSIZE 30720
unsigned long int FileSize;

/*
 This describes the byte quantities that will be moved to/from the port.
*/
typedef unsigned char byte;

/*
  Global allocation of variables used by BufferToPort and PortToBuffer.
*/
byte bLowByte = 0;
byte bHiByte = 0;
byte bCombinedByte = 0;
byte bFinishedByte = 0;

byte TransferBuffer[BUFFERSIZE];
byte FileName[256];
byte FileLength[4];

/*
 This defines the value output to base + 2 before attempting a read. This allows
 the open collector outputs of these to pins 'float' so that external logic
 voltages may be impressed on them, i.e. the TTL data from the sending computer.
*/
#define READMASK 0x04

/*
 This defines a macro to shift the byte read from base + 2 one bit to the right,
 so the result may be ORed with the byte from offset 1.
*/
#define SHIFT(a) ( ( a ) >> 1 )

/*
 This 1's complements and masks out the inverted bits to normalize them.
*/
#define INVERTED( a ) ( ( (~a ) & 0x85 ) )

/*
 This masks out the normal bits so they can be recombined with the
 normalized inverted bits.
*/
# define NORMAL(a) ( ( a ) & 0x7A )
```

Listing 17.3 *continued*

```
/*
 This returns the status of the strobe.
*/
#define STROBE(a) ( ( ( a ) & 0x01 ) )

/*
  These values are used to toggle the strobe high or low.
*/
#define STROBEHIGH 0x01

#define STROBELOW  0x00

/*
 These are typical base addresses for lpt1 and lpt2
*/
#define BASE1 0x378

#define BASE2 0x278

/*
  This should provide compatibiliy with Microsofts inp() and outp() functions
  if Microsoft does not provide these macros in CONIO.H
*/
#ifdef MSC_COMPILER
#define _STAT_DEFINED
#define inportb(port) inp(port)
#define outportb(port,out) outp(port,out )
#endif

/* End of File */
```

Chapter 18

Demystifying Interrupt-Driven Serial Communications

Russell L. Jacobs

Introduction

Why does the prospect of writing serial communications programs give even the most stout-hearted programmer the shivers? The lack of experience, coupled with the mysterious "black box" image that serial ports have enjoyed for many years, may well be the answer. Most of the material available today concerns how to use communications software, but little has attempted to de-mystify the art of programming serial communications software. There are a lot of packages available on the market today that perform communications. The majority of these are used for communicating from one computer to a host computer. The host can be a mainframe, a mini-computer, or another micro-computer. One of the most popular uses for these communications packages is connecting with the thousands of bulletin boards available for the exchange of information, ideas, and files.

With this explosion in the growth of serial communications, chances are that most programmers will have to butt heads with an `RS-232` port at some time or another, probably sooner than later. In the past, when I have been faced with the prospect of venturing into unknown territory, I have always brought a reference book as a companion. I hope that within the following pages, you will find a companion for your journey into programming serial communications.

Objectives

In the following chapter, I will explore programming the serial port, commonly referred to as the `RS-232` port. My main focus will be asynchronous serial communications. In its simplest form, asynchronous serial communications requires two ports connected by some means (by a direct cable or through telephone lines, for example). One port assumes the role of sender and the other acts as receiver. The sender takes a byte and outputs it one bit at a time; the receiver then assembles this "bit-stream" back into a byte.

I will first give a brief overview of interrupts. Then I will describe the functions available through DOS and the BIOS (basic input/output system) of MS-DOS machines. I will explore the hardware itself and different methods of controlling the `RS-232` port. Once I have gone over the hardware basics, I will cover programming the `RS-232` port in C. Finally, I will construct an interrupt-driven serial program.

While the scope of this chapter is relatively narrow, the material presented here will help you understand and tackle programming the serial port.

Interrupts

Interrupts are, in their simplest form, the computer's way of letting your program know that something has occurred and affording you the opportunity to take appropriate action.

When an interrupt occurs, the program currently running is suspended. The CPU looks at the Interrupt Vector Table for an address to a function (or interrupt handler), then branches to this function. It is the responsibility of this function to perform whatever processing is necessary for the given interrupt, then to clear the interrupt. In the case of serial interrupts, the Programmable Interrupt Controller (PIC) must also be reset. The PIC is located at address `20` hex, and the reset value is also `20` hex. The interrupt handler `Com_Handler()`, in the program `COMM.C`, resets the PIC before exiting the function.

While that may sound like a mouthful, it is really quite simple. For this discussion I will refer to the interrupts generated by activity at the serial port. There are four different types of interrupts that can be generated by the serial port (these are discussed in detail later on). You decide which of these interrupts you want the serial port to inform you of by enabling that interrupt.

Let's take the Data Received Interrupt as an example. This interrupt is generated when an incoming character has been received. You tell the serial port that you want to be informed of this event by setting bit `0` of the Interrupt Enable Register (`IER`) to `1` (this is discussed in detail later on). When a character comes into the port, an interrupt is generated. The default interrupt handler will simply read the character to clear the interrupt and discard it.

In order to override this default behavior, you write a function that will take the place of the default interrupt handler and replace the address of the default handler in the vector table with the address of your function. You can now perform any action you wish inside your function when this interrupt occurs. Using our current example, you may wish to read in the character, place it into a buffer, increment an index, and reset the PIC.

One very important item to remember is that if you receive an interrupt, be certain that at the very least you clear the interrupt.

DOS and BIOS Functions

The DOS and BIOS functions are services available to programmers. The BIOS functions are stored in ROM chips that are a part of the machine itself. These are known as ROM-BIOS functions, but for the sake of simplicity, I will refer to them as simply BIOS. The DOS functions are loaded into memory when the operating system boots-up. The two sets of functions are identical in their form and use. DOS functions will allow a greater deal of flexibility and machine independence, while some of the BIOS functions are more powerful. The DOS functions will have a consistent implementation between different versions of MS-DOS, as well as with other DOS vendors. The BIOS functions, on the other hand, are machine specific, and their implementation may not be the same on hardware produced by different manufacturers.

How DOS and BIOS Services Are Invoked

As I mentioned earlier, both DOS and BIOS functions are identical in their form and use. These services are accessed through software interrupts. How the interrupts are coded depends upon the language being used. Software interrupts are a means of branching to the location where the instructions for the service are stored, executing these services, and returning.

When the processor receives an interrupt instruction, the processor looks into memory and reads the Interrupt Vector Table. This area contains the addresses for all possible interrupts. The processor will then execute the function stored at that address and return to the calling

```
unsigned int uiSetParms (int iPortnum, unsigned int uiParms)
{
union Regs InRegs, OutRegs;

InRegs.x.dx = iPortnum;
InRegs.h.ax = 0;
InRegs.h.al = uiParms;
int86 ( 0x14, &InRegs, &OutRegs );
return ( OutRegs.x.ax );
}
```

Figure 18.1 *Accessing the DOS and BIOS functions from within a program.*

Purpose	Interrupt	Function	Description	Calling Registers	Return Registers
Auxiliary Input	21h	03h	Reads a character in from the first serial port.	AH = 03h	AL = 8-bit data to output to STDAUX
Auxiliary Output	21h	04h	Writes a character to the first serial port.	AH = 04h DL = 8-bit data to output to STDAUX	None

Table 18.1 *DOS serial communications interrupt functions.*

program via an IRET (interrupt return) instruction. The processor will then continue execution at the next line of code.

In order to access the DOS and BIOS functions from within a program, you must load the CPU's registers with the appropriate values and then invoke the desired interrupt. An example of how this is done is shown in Figure 18.1.

The example function takes two parameters, the `COM` port number (this is `0` for `COM1` and `1` for `COM2`) and the parameter byte. The `REGS` data structures `InRegs` and `OutRegs` contain the register structure of the 80x86 CPU, one for input to the CPU and the other for the return values. The `REGS` structure allows for easy access into the registers, either the full 16-bit register or the upper or lower 8 bits. The structure is broken into two substructures to accomplish this. For access into the entire 16-bit `DX` register, reference the `InRegs.x.dx` portion of the structure. To access just the upper or lower half of the same register, use `InRegs.h.dl` for the lower byte, or use `InRegs.h.dh` for the upper byte.

The example function uses the BIOS `INT 14h` to initialize the `COM` port. DOS and BIOS interrupts contain functions within them. The `AH` register is used for requesting the proper function within a given interrupt.

In the case of the example function in Figure 18.1, the function requested is `0`. If I had loaded the `AH` register with a `01h`, the example function requested would write a character to the `COM` port. In Figure 18.1, I load the `DX` register with the requested `COM` port, the `AH` register with `0h` to request initialization, and `AL` with the parameter byte specifying the configuration requested. `INT 14h` is then called by means of the C function `int86`. The status is returned in the `AX` register, the port status in the `AH` register and the modem status in the `AL` register.

The BIOS serial communications functions use `INT 14h` and have functions `0h` through `05h` associated with them. The two DOS functions use `INT 21h`, and the function numbers `03h` and `04h`. These are used for reading and writing (respectively) to a serial port. The DOS and BIOS interrupt functions are listed in Table 18.1 and Table 18.2, respectively.

In short, to use a DOS or BIOS function, you simply load the input registers with the appropriate values, call the interrupt, and take action based upon the returned register values.

Purpose	Interrupt	Function	Description	Calling Registers	Return Registers
Initialize Communications Port	14h	00h	Sets serial port parameters (baud rate, parity, stop-bits, and word length).	AH = 00h AL = Initialization parameter DX = Port number *	AH = Port status AL = Modem status
Write Character to Communications Port	14h	01h	Outputs character to serial port.	AH = 01h AL = Character to output DX = Port number *	AH = Bit 7 = 0, the function succeeded. AL = Bit 7 = 1, the function failed. Bits 0-6 show the cause of the failure.
Read Character from Communications Port	14h	02h	Inputs character from serial port.	AH = 02h DX = Port number *	AH = Bit 7 = 0, the function succeeded. AL = Character read. AL = Bit 7 = 1, the function failed. Bits 0-6 show the cause of the failure.
Request Communications Port Status	14h	03h	Returns the status information about a serial port.	AH = 03h DX = Port number *	AH = Port Status AL = Modem Status
Extended Initialization (PS/2 only)	14h	04h	Initializes serial port parameters.	AH = 04h AL = Break Setting BH = Parity BL = Stop Bits CH = Data Length CL = Transmission Rate DX = Port number *	AH = Port Status AL = Modem Status
Extended Communication Control (PS/2 only)	14h	05h	Allows extended control of the modem control register.	AH = 05h AL = Read or Write Modem Control Register (00h = read, 01h = write) BL = Modem Control Register DX = Port number *	AH = Port Status AL = Modem Status BL = Modem Control Register

Table 18.2 ***BIOS serial communications interrupt functions.***
** 0 for COM1, 1 for COM2, 2 for COM3, 3 for COM4). Note: COM3 and COM4 are only valid on AT and higher class machines.*

Hardware

Before attempting to write a serial communications program, you must have a good understanding of the underlying hardware. The INS 8250 chip is the UART (Universal Asynchronous Receiver and Transmitter) most commonly used to handle serial communications for IBM PC and compatibles.

The UART has four main responsibilities:

1. It converts parallel signals from the system's CPU into serial signals for transmission from the UART. It also performs the reverse task, taking serial input to the computer and converting it into parallel information the CPU can use.
2. It adds the necessary handshaking information to each character being transmitted and strips this information from incoming characters.
3. It computes the parity bit on all characters sent or received, ensures the characters are sent at the correct baud rate, and reports any errors that have occurred.
4. It sets up the hardware handshaking and reports on the status of incoming signals.

Like the CPU, the 8250 has registers, or memory storage locations. The 8250 has three types of registers, Control, Status and Buffer. These registers are accessed by using the `COM` port address plus an offset. The base addresses for `COM1:` through `COM4:` are shown in the Figure 18.2.

Each register can be accessed by using the base address listed above and adding the appropriate offset. Here is a description of the individual registers is listed below, along with the offsets necessary to access them:

COM1:	03F8h
COM2:	02F8h
COM3:	03E8h
COM4:	02E8h

Figure 18.2 ***Base addresses for communications ports on the IBM PC and Compatibles.***

LSR — Line Status Register (Offset 5)

All of the references I have ever seen begin with Offset 0 and move through the list. Normally this would be the logical approach. But since a number of the other registers depend directly on the status of the LSR, I will present this one first.

The LSR allows you to monitor the status of the communication line. Any change to the state of the LSR generates an interrupt, and will allow your interrupt routine to take the appropriate action. Listed in Figure 18.3 are the individual bits of the LSR and their meanings.

THR — Transmitter Holding Register (Offset 0)

The THR is a write only register used to hold a byte of data that is about to be sent. As soon as the UART transmitter is available, the character is sent and this register is free to accept another byte for transmission. You can write to this register when bit *5* of the LSR (described above) is clear.

All meanings are for bit values of 1.

BIT 0:	Data Ready. This bit indicates that there is a byte in the Receiver Data Register (RCR). This bit is reset when the byte is read.
BIT 1:	Overrun Error. This bit indicates that a character was received prior to the last one being read. The first byte is lost and overwritten by the new incoming byte.
BIT 2:	Parity Error. An incoming byte's parity bit is wrong. This bit will never be set if no parity is being used. This bit is reset when the byte is read.
BIT 3:	Framing Error. The incoming byte does not have a valid stop bit. This bit is reset after the byte is read.
BIT 4:	Break Detect. If the input line is held low for more than the maximum character length. This bit is reset when read.
BIT 5:	THR Empty. The Transmitter Holding Register (THR) is empty. The UART is ready to receive another character for transmission. If you do not check this bit, you may overwrite a character awaiting transmission.
BIT 6:	TSR Empty. Transmitter Shift Register (TSR) is empty. This bit will indicate that all bytes have been transmitted. Check this bit to ensure that all bytes have been transmitted. Failure to do so could result in incomplete transmission.
BIT 7:	Spare. Always 0.

Figure 18.3 *Bits of the 8250 Line Status Register.*

RDR — Receiver Data Ready (Offset 0)

The RDR is a read-only register that holds the byte most recently received from the communications line. You read this register when bit *0* of the LSR indicates that a character is waiting. Failure to process a character can result in it being overwritten by the next incoming character.

IER — Interrupt Enable Register (Offset 1)

The IER is used to tell the 8250 which events should generate an interrupt. There are four types of interrupts that you can enable. You should note, however, that if you enable an interrupt, you must act on that interrupt in order to clear the interrupt. For example, if you enable bit *3* of the IER, interrupts will occur when there is a change in the MSR (Modem Status Register). You can clear this interrupt by simply reading the MSR. Of course, if you enabled this interrupt, you would probably want to do more than just clear it.

IIR — Interrupt Identification Register (Offset 2)

When an interrupt occurs, your interrupt-driven program reads this register to determine what caused the interrupt. There are five values in the IIR. Four correspond to the interrupt types, while the fifth is used as

All meanings are for bit values of 1.

BIT 0:	Data Received. This interrupt is generated whenever a character has been received and moved into the RDR.
BIT 1:	THR Empty. This interrupt will occur whenever the THR is empty. This will signal your routines that the UART can accept another byte for transmission.
BIT 2:	Receiver Line Status. This indicates an error condition on the line. The errors can be obtained from the Line Status Register (LSR). The possible error conditions are parity, overrun, framing and break.
BIT 3:	Modem Status. This interrupt will be generated when there is a change in any of the incoming modem status lines. The Modem Status Register (MSR) should be read and appropriate action taken. This interrupt should be enabled if you are using CTS or DSR for handshaking.
BITS 4 - 7:	Not Used. Always 0.

Figure 18.4 ***Bits of the 8250 Interrupt Enable Register.***

a flag to let you know if more interrupts are pending. The possible values are listed below in Figure 18.5.

LCR — Line Control Register (Offset 3)

This register is the primary control line for the UART. It controls how the UART sends and receives information.

```
All meanings are for bit values of 1.

BIT 0:        No Interrupts pending.
BIT 1-2:      Interrupt ID.
              00 - Change in MSR
              01 - THR Empty.
              10 - Data Received
              11 - Receiver Line Status Error.
```

Figure 18.5 *Bits of the 8250 Interrupt Identification Register.*

```
BIT 0-1:      Character Length
              00 - 5 Bits
              01 - 6 Bits
              10 - 7 Bits
              11 - 8 Bits
BIT 2:        Stop Bits
              0 - 1 Bit
              1 - 1.5 Bits  (if using 5-bit characters)
              1 - 2 Bits    (if using 6,7 or 8 bit characters)
BIT 3-5:      Parity
              000 - Ignore (No Parity)
              100 - Odd Parity
              110 - Even Parity
              101 - Mark
              111 - Space Parity
BIT 6:        Break Condition. This is a 1 if enabled, otherwise it is
              set to 0.
BIT 7:        Divisor Latch Access. This bit allows access to the LSB
              and MSB Baud Rate Divisor Latch (BRDL) registers. Setting
              this bit to 1 disables the normal states of registers 0
              and 1 and converts them to the LSB and MSB BRDL registers
              respectively. This bit must be set to change the baud
              rate. It must also be  set back to 0 for normal operation
              of registers 0 and 1.
```

Figure 18.6 *Bits of the 8250 Line Control Register.*

MCR — Modem Control Register (Offset 4)

The MCR controls the handshaking signals sent out from the UART. There are five bits that can be set or cleared by your program.

MSR — Modem Status Register (Offset 6)

The MSR gives information about the state of the four status lines of the modem. There are two bits for each of these four lines. Bits *1* to *4* are delta bits. They will indicate if there is has been a change since the last time the register was read. Figure 18.8 details the MSR.

SCR — The Scratch Register (Offset 7)

This register has no function and is generally not used. It can be used as a byte of RAM, but early versions of the 8250 did not have this register.

BRDL — Baud Rate Divisor Latch (Offset 0 - LSB, Offset 1 - MSB)

You may remember that the BRDL registers were mentioned earlier. And you may also have noticed that the offsets to these registers have been used for other registers. The way this is accomplished is by using the DLAB bit of the Line Control Register (LCR). By setting the DLAB to

BIT 0:	Data Terminal Ready (DTR). This bit is used to inform a remote device of our current status. If this bit is 0, a remote device can send data. If it's a 1, the remote device should not send data.
BIT 1:	Request To Send (RTS). When this bit is 1 it tells the modem that the computer is ready to send data to the line.
BIT 2:	Out1. User-defined output. This bit has no function on IBM-PC serial cards.
BIT 3:	Out2. User-defined output. If this bit is not set to 1 no interrupts can occur. It is necessary to leave this bit set to 1 to enable interrupts.
BIT 4:	Local Loopback. This bit will allow you to perform diagnosis without a loopback connector. When this bit is set, all data written to the UART will be read back in (or looped back) for easy testing and debugging on one system.
BIT 5-7:	Not Used. Always 0.

Figure 18.7 ***Bits of the 8250 Modem Control Register.***

a *1*, registers `0` and *1* are disabled, and all input and output requested at their addresses will now access the BRDL registers. Make certain that if you change the DLAB in the LCR to a *1*, you reset it to resume normal operations of registers `0` and *1*.

Programming the RS-232 Port in C

So far I have covered DOS and BIOS functions, the 8250 UART, its registers, and their meanings. Now we need to put this knowledge to work.

Listing 18.1 (`COMM.C`) is a C source code file that demonstrates interrupt-driven serial I/O. The easiest place to start is to `#define` the port address, register addresses, and other values needed for initializing and communicating with the port. The header file `COMM.H` (see Listing 18.2) handles the most common of these defines. As a matter of programming style, I prefer to use `inp( ComPort );` rather than `inp( 0x2f8 );` just for readability and maintainability of the code.

Please note that the program was written to use `COM2` (my mouse got extremely cranky when I tried to use `COM1` at the same time he was!). If you want to use this program on `COM1`, simply change `ComPort` from `0x2f8` to `0x3f8`, and `IRQ` from `3` to `4` in `COMM.H`. The other change required is in the function call `ComPort_Init()` in the `COMM.C` (Listing 18.1) file. Change the first parameter from a *1* to a `0` and recompile. The program was

All meanings are for bit values of 1.

BIT 0:	Delta Clear To Send (CTS). The CTS line has changed.
BIT 1:	Delta Data Set Ready (DSR). The DSR line has changed.
BIT 2:	Delta Trailing Edge Ring Indicator (TERI). This bit is set high on the trailing edge of the ringer pulse from the modem.
BIT 3:	Delta Data Carrier Detect (DCD). This bit is set high when the modem is connected to another modem.
BIT 4:	Clear To Send. The CTS input is high, indicating that it is OK to send data.
BIT 5:	Data Set Ready. The DSR input is high, indicating that there is data ready in the modem.
BIT 6:	Ring Indicator. The ring indicator is high, indicating the phone is ringing.
BIT 7:	Data Carrier Detect. The DCD line is set high, indicating a connection to another modem.

Figure 18.8 *Bits of the 8250 Modem Status Register.*

compiled using Microsoft C 6.0 using the small memory model (to compile : `cl /AS comm.c`).

If you look through `COMM.H` you will see all the defines, structures, and function prototypes. Let's first look at the defines. I have defined the values for the Baud Rate, Parity, Wordlength, and Stop Bits. These values will be `or`'d together to create a parameter byte to initialize the chosen `COM` port. `QueueSize` represents the size of the input and output queues. This value makes programming fairly simplistic. By using an unsigned `char` as the index, rollover is automatic when you exceed the 256 number limit. If you want to further enhance this buffering scheme, and I'm certain you will if you intend to create an industrial strength application, you will have to rewrite the indexing and buffering methods.

The next values are the `IRQ` (interrupt request) and the `ComPort` variables. This program was written simply to deal with one com port. This can easily be expanded by creating a structure containing the information for the com port and initializing this structure during `com` port initialization.

You will notice quite a few TLAs in the following text. (TLA is the ultimate in redundancy... it's a Three Letter Acronym for Three Letter Acronym). These represent the registers of the 8250. I have also defined `IER_RECIEVE` and `IER_TRANSMIT`. These values represent the bits of the Interrupt Enable Register for Receiving and Transmission interrupts to be enabled.

The remaining defines are the interrupt IDs available on the 8250 UART. These values will be used within the interrupt handler to determine the cause of the interrupt.

The only structure used is called `QUEUE`. It contains a 256-character buffer, a write and read index. Two variables of this type are declared globally, `InQueue` and `OutQueue`. The remaining variables are to hold the old values of the Interrupt Vector Table to restore the machine to its original state before the program returns to DOS.

The remainder of the file contains function prototypes.

When looking at the code, I find it easiest to begin with `main()` and work my way down (just like the system will). The only variable `main()` uses is `Key`. `Key` holds the characters typed at the keyboard before transmitting them to the UART and the characters received when printing them to the screen.

The Input and Output buffers indexes are cleared first. The next step is to install my own interrupt handler in place of the default one. This is done in the function `Install_Handlers()`. The original values stored in the vector table for the serial interrupt handler and the Control-C handler are saved and replaced with pointers to the functions `Com_Handler()` and `CtrlC_Handler()`, respectively. `Com_Handler()` deals with the serial interrupts, and the `CtrlC_Handler()` intercepts the *Ctrl-C* to prevent the program from being terminated abnormally. (See Chapter 10 for industrial-strength *Ctrl-C* trapping.)

The `ComPort_Init()` function takes the port number, baud rate, parity, word length and stop bits as parameters and initializes the serial port using the BIOS function `14h` described earlier. The last several lines of the function use the C function `outp()` to set the bit values of the Modem Control Register and Interrupt Enable Register to enable interrupts. The last command, `inp(ComPort)`, simply clears out any data that may be in the register.

The heart of the program takes place in the `for(;;)` loop. There are three functions that handle all of the communications interrupts. The `Write_Char()` function takes a character from the keyboard and puts it in the output queue then makes sure that the transmit interrupt is enabled. The `Read_Char()` function retrieves a waiting character if there is one. These two functions depend on the `Com_Handler()` function.

The IIR register is read to determine which interrupt occurred. The `MODEM_STATUS` and `LINE_STATUS` interrupts should not occur since they have not been enabled in the IER, but code to clear the interrupt has been put in as an example of how to handle them.

The interrupts to be concerned with are the `TRANSMIT` and `RECIEVE` interrupts.

To handle the `TRANSMIT` interrupt, the queue buffer is checked to see if there are more characters to be transmitted. If there are not, then the transmit interrupt is disabled. Otherwise the character is output to the port.

The `RECIEVE` interrupt is handled in very much the same fashion. First the character is read in, and if there is room in the input buffer, it is placed into the queue. Otherwise it is discarded.

The default action is to just reset the PIC (Programmable Interrupt Controller). This action is also done each time through the interrupt handler, just before the interrupts are enabled again.

The final actions in the `main()` function are simply to clean-up prior to exiting. The MCR is set to `0`, to disable interrupts and clear the RTS and DTR bits. Finally the function `Restore_Handlers()` is called to restore the original interrupt handlers saved in `Install_Handlers()` to the vector table prior to exiting.

Summary

Over the years, computers have changed quite a bit. As I've witnessed the evolution of personal computers, one component has remained a virtual constant, the UART. If you look at the bibliography for this chapter, you'll notice the dates for my references are from 1986 to 1988. Any reference you can find on programming the serial port is well worth hanging onto.

The program in this chapter has been designed to show the basics of serial port communications. To really exploit the abilities of the serial port you will have to stretch yourself and be prepared to lock up the system more than once. I hope I have been able to give you a push in the right direction.

Bibliography

DOS Programmers Reference — Terry R. Dettmann 1988 QUE Corporation, Carmel Indiana.

C Programmers Guide to Serial Communications — Joe Campbell 1987 Howard W. Sams & Company, Indianapolis, Indiana.

Mastering Serial Communications — Peter W. Gofton 1986 Sybex Inc., Almeda California

Listing 18.1 COMM.C

```
/**************************************************
;
;  Author      : Russell L. Jacobs
;                Copyright 1993, SoftWare Alchemy
;
;  Date        : 11-28-93
;  Description : This is the C program file for interrupt-driven serial
;                input-output featured in MS-DOS System Programming, 3rd ed.
;
;  Compile Options : Written and Compiled using
;                    Microsoft C 6.0.
;                    To compile: C:\>cl /AS comm.c
;
;  NOTE : To compile this program under Borland C, the inp()
;         and outp() function calls need to be replaced by
;         Borlands inportb() and outportb() calls respectively.
;         This could be done with a #ifdef statement.
;
**************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
#include <string.h>
#include <conio.h>

#include "comm.h"

/****************************************************
;  Function : static void interrupt far Com_Handler()
;
;  Notes    : This function is not directly called from any other
;             function. The address of this function is inserted into
;             the interrupt vector table to overwrite the default
;             interrupt handler and process the serial port interrupts.
;
;  Inputs   : None
;  Outputs  : None
;
****************************************************/
static void interrupt far Com_Handler()
{
  int Ch;

/* disable interrupts while processing this one */
_disable();
```

Listing 18.1 *continued*

```
   /* check the Interrupt ID Register */
 switch (inp (IIR) )
   {
   /* Just read the MSR to clear it */
   case MODEM_STATUS:
     inp (MSR);
     break;

   case TRANSMIT:
     /* if no other chars are waiting in buffer disable the    */
     /* transmit interrupt, otherwise output the character     */

     if ( OutQueue.Read_Index == OutQueue.Write_Index)
          outp ( IER, IER_RECIEVE );
     else
      {
       Ch = OutQueue.Queue [ OutQueue.Read_Index++ ];
       outp ( THR, Ch );
      }
      break;

   case RECEIVE:
     /* Read in character and place it into input buffer,  */
     /* if buffer is full, discard the new character.       */

     Ch = (unsigned char) inp ( RDR );
     if ((InQueue.Write_Index+1) != (InQueue.Read_Index))
          InQueue.Queue[ InQueue.Write_Index++] = Ch;
     break;

   /* just read the LSR to clear it */
   case LINE_STATUS:
     inp ( LSR );
     break;

   default:
     outp( 0x20, 0x20 );  /* Reset Pic */
     break;
      }

outp( 0x20, 0x20 );            /* Reset Pic */
_enable();                     /* Enable interrupts */
}
```

Listing 18.1 *continued*

```
/*****************************************************
;  Function : int ComPort_Init ( int PortNum,
;                                int BaudRate,
;                                int Parity,
;                                int WordLength,
;                                int StopBits )
;
;  Notes    : This function is called to initialize
;             the com port with the attributes passed
;             into it.
;
;  Inputs   : PortNum    - 0 = Com1, 1 = Com2
;             BaudRate   - BAUD300, BAUD1200, BAUD2400,
;                          BAUD4800 or BAUD9600
;             Parity     - NOPARITY, ODDPARITY
;                          or EVENPARITY
;             WordLength - LEN7 or LEN8
;             StopBits   - STOP1 or STOP2
;
;  Outputs  : 0 = Successful ComPort Initialization
;             1 = Invalid Baud Rate
;             2 = Invalid Parity
;             3 = Invalid Word length
;             4 = Invalid Stop bits
;
*****************************************************/
int ComPort_Init ( int PortNum, int BaudRate, int Parity,
                   int WordLength, int StopBits )
{

   union REGS InRegs, OutRegs;
   unsigned char Parms = 0;

/* disable interrupts during initialization */
_disable();

   /* Or the Baud Rate into parm byte */
   switch (BaudRate)
      {
      case 300:
        Parms |= BAUD300;
        break;

      case 1200:
        Parms |= BAUD1200;
        break;

      case 2400:
        Parms |= BAUD2400;
        break;
```

Listing 18.1 *continued*

```
   case 4800:
     Parms |= BAUD4800;
     break;

   case 9600:
     Parms |= BAUD9600;
     break;

   default: return ( 1 );
   }

/* Or the Parity into parm byte */
switch (Parity)
   {
   case NOPARITY:
     Parms |= NONE;
     break;

   case ODDPARITY:
     Parms |= ODD;
     break;

   case EVENPARITY:
     Parms |= EVEN;
     break;

   default: return ( 2 );
   }

/* Or the WordLength into parm byte */
switch (WordLength)
   {
   case 7:
     Parms |= LEN7;
     break;

   case 8:
     Parms |= LEN8;
     break;

   default: return ( 3 );
   }

/* Or the Stop Bits into parm byte */
switch (StopBits)
   {
   case 1:
     Parms |= STOP1;
     break;
```

Listing 18.1 *continued*

```
      case 2:
        Parms |= STOP2;
        break;

      default:return ( 4 );
      }

/* Initialize the port using the BIOS function */
InRegs.h.ah = 0;
InRegs.x.dx = PortNum;
InRegs.h.al = Parms;
int86 ( 0x14, &InRegs, &OutRegs );

/* set RTS and DTR */
outp( MCR, 11 );

/* Enable Trans/Recieve interrupts */
outp( IER, IER_RECIEVE | IER_TRANSMIT );

/* clean out RDR   */
inp ( ComPort );

/* enable interrupts */
_enable();

return( 0 );
}

/*****************************************************
;  Function : void Install_Handlers()
;
;  Notes    : This function saves the original vector table entry
;             of the com port interrupt handler, sets the vector
;             table entry to point to the function Com_Handler. The
;             Control-C interrupt is also saved and replaced, to
;             prevent the program from being terminated abnormally.
;
;  Inputs   : None
;  Outputs  : None
;
*****************************************************/
void Install_Handlers()
{
  int SerialVector = IRQ+8;

  /* Save the old serial interrupt handler */
  Old_Serial_Handler = _dos_getvect( SerialVector );

  /* Set the vector table to point at Com_Handler() */
  _dos_setvect( SerialVector, Com_Handler );
```

Listing 18.1 *continued*

```
  /* Save the old Control-C Handler */
  Old_CtrlC_Handler = _dos_getvect( CTRLC );

/* Set the vector table to point at CtrlC_Handler() */
  _dos_setvect( CTRLC, CtrlC_Handler );
}

/***************************************************
;  Function : void Restore_Handlers()
;
;  Notes    : This function is the counterpart to
;             Install_Handlers(). It will restore the
;             default state of the serial port handler
;             and the Control-C handlers in the vector
;             table.
;
;  Inputs   : None
;  Outputs  : None
;
***************************************************/
void Restore_Handlers()
{
  int SerialVector = IRQ+8;

  /* Restore the original interrupt handlers */
  _dos_setvect( SerialVector, Old_Serial_Handler );
  _dos_setvect( CTRLC, Old_CtrlC_Handler );
}

/***************************************************
;  Function : void interrupt far CtrlC_Handler()
;
;  Notes    : This function does nothing except prevent
;             a Control-C from terminating the program.
;             This is not the cleanest way to intercept
;             and handle this interrupt, but you can
;             refer to chapter 10 for a detailed
;             discussion on Control-C handling.
;
;  Inputs   : None
;  Outputs  : None
;
***************************************************/
void interrupt far CtrlC_Handler()
{
return;
}
```

Listing 18.1 *continued*

```
/*****************************************************
;  Function : int Write_Char ( unsigned char Ch )
;
;  Notes    : This function will put a character into
;             the output queue if it is not full, then
;             make sure transmit interrupts are
;             enabled.
;
;  Inputs   : unsigned character Ch to be output
;  Outputs  : Success returns the original character
;             passed in.
;             -1 = Queue full, character discarded
;
****************************************************/
int Write_Char ( unsigned char Ch )
{

 if ((OutQueue.Write_Index+1) == OutQueue.Read_Index)
      return ( -1 );
 OutQueue.Queue [ OutQueue.Write_Index ] = Ch;
 OutQueue.Write_Index += 1;
 if (( inp ( IER ) & IER_TRANSMIT ) == 0)
    outp ( IER, IER_TRANSMIT | IER_RECIEVE );
 return ( Ch );

}

/*****************************************************
;  Function : int Read_Char ( void )
;
;  Notes    : This function will put a character into
;             the output queue if it is not full, then
;             makes sure transmit interrupts are
;             enabled.
;
;  Inputs   : None.
;  Outputs  : Success returns the next available
;             character from the input Queue.
;             -1 = No characters available.
;
*****************************************************/
int Read_Char ( void )
{
   if (InQueue.Write_Index == InQueue.Read_Index)
      return( -1 );
   else
      return(InQueue.Queue [ InQueue.Read_Index++ ]);
}
```

Listing 18.1 *continued*

```
/*****************************************************
;  Function : int main ( void )
;
;  Notes    : This is the main function in COMM.C.
;
;  Inputs   : None
;  Outputs  : 0 = Success
;             1 = Error in ComPort Initialization
;
*****************************************************/
int main ( void )
{
  int Key = 0;  /* Character read in from keyboard */

/* clear the input and output queue indexes */
  InQueue.Write_Index  = 0;
  InQueue.Read_Index   = 0;
  OutQueue.Write_Index = 0;
  OutQueue.Read_Index  = 0;

/* install the interrupt handlers */
  Install_Handlers();

/* initialize the Comport */
  if ((ComPort_Init(1, 2400, NOPARITY, 8, 1)) !=0)
   {
    printf("%s\n", "Error in comport initialization");
    return ( 1 );
   }

  for (;;)       /* loop until ESC pressed */
     {
     if (kbhit())
      {
         Key = getch();     /* get key */
         if (Key == 27)
           break;           /* quit if ESC */
         else
           /* send key to port */
           Write_Char ( (unsigned char)Key );
      }
/* if a char has been recieved, read & display */
     Key = Read_Char ();
     if ( Key >= 0 )
        {
         putc ( Key, stdout );
        }

     }
```

Listing 18.1 *continued*

```
  outp( MCR,  0 );    /* clear RTS and DTR */

/* restore the original interrupt handlers */
  Restore_Handlers();

return (0);

}

/* End of File */
```

Listing 18.2 COMM.H

```
/***************************************************
;
; Author      : Russell L. Jacobs
;               Copyright 1993, SoftWare Alchemy
;
; Date        : 11-28-93
; Description : This is the Header file for Comm.c.
;               It contains all defines, structure
;               definitions, and prototypes for the
;               interrupt driven serial
;               communications program in MS-DOS
;               System Programming, 3rd edition.
;
***************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
#include <string.h>
#include <conio.h>

#define BAUD300     0x40    /* Baud Rate Defines */
#define BAUD1200    0x80
#define BAUD2400    0xA0
#define BAUD4800    0xC0
#define BAUD9600    0xE0

#define NOPARITY    0       /* Parity Type Defines */
#define ODDPARITY   1
#define EVENPARITY  2

#define NONE        0x00    /* Parity Defines */
#define ODD         0x18
#define EVEN        0x8

#define LEN7        0x02    /* Word Length Defines */
#define LEN8        0x03

#define STOP1       0x00    /* Stop Bit Defines */
#define STOP2       0x40

/* Size of Input/Out Buffers */
#define  QueueSize     256
/* 0x3F8=COM1, 0x2F8=COM2    */
#define  ComPort     0x2F8
/* 4=COM1, 3=COM2            */
#define  IRQ           3
```

Listing 18.2 *continued*

```
/* Transmitter Holding Register Address      */
#define THR   ComPort+0
/* Receiver Data Register Address            */
#define RDR   ComPort+0
/* Interrupt Enable Register Address         */
#define IER   ComPort+1
/* Interrupt Identification Register Address */
#define IIR   ComPort+2
/* Line Control Register Address             */
#define LCR   ComPort+3
/* Modem Control Register Address            */
#define MCR   ComPort+4
/* Line Status Register Address              */
#define LSR   ComPort+5
/* Modem Status Register Address             */
#define MSR   ComPort+6

/* To Set the Recieve interrupt              */
#define IER_RECIEVE    1
/* To Set the Transmit interr

/* Ctrl-C interrupt number                   */
#define CTRLC 0x23

/* Modem Status Interrupt ID                 */
#define MODEM_STATUS 0
/* Transmit Interrupt ID                     */
#define TRANSMIT     2
/* Recieved Data Interrupt ID                */
#define RECEIVE      4
/* IIR Line Status Interrupt ID              */
#define LINE_STATUS  6

/* Structures */

/* Input/Output Queue structure */
typedef struct _QUEUE {
    char Queue[QueueSize];
    unsigned char Write_Index;
    unsigned char Read_Index;
    } QUEUE;
```

Listing 18.2 *continued*

```
/* Global Data Definitions */

/* Input and Output Queue structures  */
QUEUE InQueue;
QUEUE OutQueue;

/* old serial vector address */
void (interrupt far * Old_Serial_Handler)();

/* old ctrl-c vector address */
void (interrupt far * Old_CtrlC_Handler)();

/* Function Prototypes */

/* Interrupt routine        */
static void interrupt far Com_Handler( void );
/* Comport init routine     */
int ComPort_Init ( int, int, int, int, int );
/* Int. Handlers install    */
void Install_Handlers( void );
/* Int. Handlers restore    */
void Restore_Handlers( void );
/* Control C Handler        */
void interrupt far CtrlC_Handler( void );
/* Write char to port       */
int Write_Char ( unsigned char );
/* Read Char from port      */
int Read_Char  ( void );
/* Main routine             */
int main ( void );

/* End of File */
```

Chapter 19

VGA Animation Graphics

Rick Ladymon

Introduction

The Virtual Graphics Adapter (VGA) video display card is one of the most popular video display cards. This chapter will concentrate on how to perform graphic animation on the VGA. I will first introduce some of the concepts and techniques of VGA programming, then I will show you how to apply those techniques with a complete sample graphics program.

Video Display Modes

There are several different types of VGA monitors and display cards. For each display card there are also several different video display modes. The major differences between the display modes are how many pixels can be displayed at one time and how many colors are available to display them in. I will focus my discussion on VGA display mode 16, which is supported by all VGA monitors and display cards. For a more complete discussion of the different display modes available to the VGA, consult a book on PC display graphics.

The Screen

When the VGA is in display mode 16, there are 640 pixels across the screen and 350 pixels down the screen, making a total of 224,000 pixels displayed on the screen at a given time. A scan line is one horizontal line across the screen, so the VGA displays 350 scan lines with 640 pixels per scan line.

Think of the pixels on the screen as a grid, where (0,0) is the top left corner of the screen and (349,639) is the bottom right corner.

A 64Kb memory block at address `A000:0000` contains the information needed to display the pixels on the screen. Each byte of video display memory contains information for eight horizontally-consecutive pixels. Since there are 640 pixels across the screen, 80 bytes are needed to hold the information about the pixels of the first scan line. So, the first 80 bytes located at `A000:0000` are used to display the first scan line (top of the screen), the next 80 bytes are for the second scan line, and so on for all 350 scan lines of the screen. The video display memory can also be thought of as a grid. There are 80 bytes per scan line and 350 scan lines.

When writing to the VGA, it is important to understand that you are always writing eight pixels worth (one byte) of information. For example, to change pixels (0,0) through (0,7), the first eight pixels on the screen, you would write data to the memory located at `A000:0000`. To change pixels (0,8) through (0,15), the next eight pixels, you write data to the memory located at `A000:0001`. If you want pixels (0,0) and (0,7) to be turned `ON` and pixels (0,1) through (0,6) to be turned `OFF`, you would write the following bit pattern (10000001), each bit representing a pixel, to the memory located at `A000:0000`. This bit pattern sets the first and last pixels in the byte to `ON` and the pixels in between to `OFF`. By writing specific bit patterns to specific bytes in video memory the programmer can effectively draw to the detail of a single pixel.

The screen needs four pieces of pixel data to construct the 16-color VGA palette. The VGA is organized into four 64Kb *color planes*, each one of which is a complete rendering of the display in one of four independent color components: blue, green, red, and intensity (light or dark). To draw a picture on the screen, you must write data to all four color planes. (Plane 0 is blue, plane 1 is green, plane 2 is red, and plane 3 is intensity). By lighting the bits in the four color planes, you can create the 16 color combinations, i.e., 0000 Black, 0001 Dark Blue, 0010 Dark Green, 0100 Dark Red, 1111 White, and so on.

You change the pixel data in the color planes by writing new data to the 64Kb video display memory at `A000:0000`. The port address `0x03C4` controls which color plane will receive the data written to video display memory. If you want to write data to a color plane you must first write the color plane number to port address `0x03C4`, where plane 0 (Blue) is `01h` (`0001b`), plane 1 (Green) is `02h` (`0010b`), plane 2 (Red) is `04h` (`0100b`), and plane 3 (Intensity is `08h` (`1000b`). The color plane whose color-plane number is stored in port address `0x03C4` will then receive all subsequent write operations until the value in port address `0x03C4` changes. Write operations to video display memory are generally specified with an offset from the first memory address at `A000:0000`. Video display offsets are primarily used in this chapter as the beginning offset for a bitmap, such as the 15 ball discussed later in the chapter. The starting screen location of the ball is calculated, then the 15 ball is written to its corresponding location in video memory.

Page Flipping

The screen can only display 28,000 bytes worth of information at one time, but the video display memory block is 64Kb. That leaves a lot of bytes left in video memory that are not being displayed to the screen at a given time. The programmer can put this space to good use using an important technique called page flipping.

Page flipping divides the video display memory into two "pages." Page 0 starts at offset 0 and page 1 starts at offset 28,000. While page 0 is displayed, the program prepares the page 1 space "behind the scenes." It then displays the page 1 screen by writing the page 1 offset to port address `0x3D4`. Meanwhile, the program prepares the page 0 space "behind the scenes." By alternating the offsets, the program can toggle indefinitely between the two pages. Page flipping allows for faster graphics and better synchronization with the scan rate.

Smoothing Out the Display

There are 350 video display scan lines, each containing 80 bytes (eight pixels per byte) on the screen. The programmer does not usually need to address the problem of updating the scan lines because this is being done for you by the VGA card. The one time that you do need to address the problem is when you are writing something that will be displayed to the screen. If the programmer changes the bytes that are currently displayed on the screen, a ripple effect can occur because the video display is out of sync with the changing of the display. (The interrupt used to constantly update the screen could be —and most probably is— somewhere in the middle of updating the screen. If you write an object to the screen, such as the pool ball, and the display update is currently updating a scan line you are changing, some of your changes will immediately take place but the ones written before will have to wait for the next vertical sync.) The VGA card comes with the ability to determine when the next vertical sync will occur. By monitoring and waiting for the next sync, then "flipping" the page, the programmer can create very smooth animation.

The VGA port address used to monitor the vertical sync pulse is `0x03DA`. The program queries this port and waits for a vertical sync pulse. When the pulse happens, the VGA has finished updating the screen and is about to start at the top again. The "page flip" should occur at this time. This keeps the screen display smooth because the data being displayed on the screen is in sync. This technique is also used to assure that the animation is smooth for all video display screens.

Example Pool Program

A simple pool game will show you a working example of graphic animation. The display begins with one cue ball and fifteen pool balls on a dark green background. The pool balls will start in their standard regulation positions. The cue ball will move forward until it strikes the rack and scatters the balls. All of the balls will move along a set pattern and reform the rack at the end of the pattern. The same sequence will continue until the user hits a key.

This simple premise will demonstrate the basic techniques of graphic animation. I wrote the code for this chapter in assembler because assembler is the fastest way to send instructions to the video display card. Some parts of the sample program would have performed nearly as efficiently in C or C++, but I wanted to maintain language consistency so I kept with assembler throughout.

The code used in this chapter has been divided into three separate files, `POOL.ASM` (Listing 19.1), `EQUATES.INC` (Listing 19.2), and `BALLS.INC` (Listing 19.3). The `POOL.ASM` file contains the source code that performs the VGA graphic animation. The `EQUATES.INC` file contains the equate data used throughout the `POOL.ASM` file. `BALLS.INC` contains the variable information used to define the pool balls and their paths. (The printed version of `BALLS.INC` has been abbreviated, but the complete listing is available on the companion code disk.)

First, take a look at Listing 19.1 `POOL.ASM`. This file begins by setting the size of the stack and including the files `EQUATES.INC` (Listing 19.2) and `BALLS.INC` (Listing 19.3). It then creates a macro called `SETREG`, which will access the VGA ports. `SETREG` is passed two arguments, the first is the VGA port address to write to, the second is the data being written to the port. `SETREG` moves the data in the `AL` register to the `AH` register, then moves the second argument passed to the `AL` register. This data is a VGA command such as, writing to a plane or changing the starting video display. (The `AL` register is set to some program-specific value before the macro is called, such as a color plane or a video display address.)

Next is the actual source code for the pool program. First, the video display mode is set to VGA graphics mode. This is accomplished by setting `AX` to `10h` and issuing an `INT 10h` function call. This interrupt function changes the screen display mode to VGA display mode 16.

The `ES` register, which I will use to point at video memory, is initialized to `A000h`, the video display memory address.

Once the display mode is set to VGA mode, the screen is cleared to dark green. To do this the program must first issue a call that tells the video display card to enable the green video plane for writing. This is accomplished by setting `AL` to `02h`, then calling the `SETREG` macro with the `VGA_INDEX` address. The `VGA_INDEX` constant is defined in the `EQUATES.INC` file as `0x03C4`, and the `WRITING` constant is defined as `2`. By calling the `SETREG` macro with the `VGA_INDEX` and `WRITING` parameters you are telling the VGA that a write is about to take place on a VGA color plane specified in the `AL` register. (The `AL` register is set to `2`, to indicate the Green color plane, before the call is placed to the `SETREG` macro). After the `SETREG` macro has completed, the VGA expects that the next write operation in video memory will affect the Green color plane. The values passed to the `SETREG` macro for the four color planes are as follows: `01h` (`0001b`) is the Blue plane, `02h` (`0010b`) is the Green plane, `04h` (`0100b`) is the Red plane, `08h` (`1000b`) is the Intensity plane, and `0fh` (`1111b`) is all of the planes. By setting `AL` to the desired plane(s) and issuing the macro call, the programmer can change the active color plane that the video memory will write to.

I then assign `DI` to point at the beginning of video memory, where `ES:DI` (`A000:0000`) is the address of the video memory to be changed. The `CX` register is set to 56,000, the number of video display bytes to change. (Since there are 28,000 bytes per display screen and I am going to flip between two pages of video memory, I need two full screens worth of video memory.)

The video display memory space is 64Kb, and I only need 56Kb to store two full screens worth of video memory. That means there is still about 9Kb worth of unused video memory. This area can be used to store objects instead of using real memory to hold the data for these objects. This technique is not used in this chapter but it is something to keep in mind. The data for objects could be read in from disk and stored in an unused area of video memory.

Next, the bit pattern to write to the VGA memory address is put into the `AL` register. The bit pattern `0FFh` (`11111111b`) represents the eight different pixels associated with this byte in video memory. Since we are clearing the screen in dark green (both pages), the bits are all set to `ON`. The REP instruction is used to store the bit pattern into each byte of the video display memory for 56,000 bytes.

Now that the screen has been cleared to dark green I am ready to begin animation of the pool balls. The data for each pool ball is defined in the include file `BALLS.INC` (Listing 19.3). Each pool ball is represented in an array two bytes wide and 16 bytes high for each of the four color planes. The details of how each pool ball is defined can be determined by studying the bit patterns found in `BALLS.INC`. I will give a more detailed explanation of the data defined in this file later when I talk about how to display the pool ball.

POOL.ASM — *The Animation Loop*

The animation loop contains the logic necessary to perform graphic animation on the pool balls. The *BX* register is used as an index to the next pool ball to be animated. The function *EachBall* displays each one of the pool balls on the screen. I will discuss the *EachBall* function in more detail later in the chapter.

To prevent screen flicker, the program must wait for the vertical sync. The function *WaitForSync* is called and doesn't return until the vertical sync has occurred. If you don't wait for the vertical sync — comment out this function call and see what happens — you will get a very uneven screen display. I'll also discuss *WaitForSync* in more detail later in the chapter.

After waiting for the vertical sync, the program then toggles which video display page is to be displayed. This is done by checking the *CurrentPage* variable. It changes from *0* to *1* or from *1* to *0*. Based on its status, the current display page, saved in the byte *POffset*, is set to either *P0_OFFSET* or to *P1_OFFSET*. As stated before, the screen is 80 bytes wide and 350 bytes high, making 28,000 bytes. The equate *P0_OFFSET* is defined as *0*, the equate *P1_OFFSET* is defined as the *SC_WIDTH* (80) multiplied by the *SC_HEIGHT* (350), so it is 28,000. The *POffset* variable is used to determine which part of video memory will receive the update.

This completes the first part of the animation process. To allow for a simple user interface, the program will check for a keystroke. If the user enters a keystroke, the program ends and the video mode is set back to text. If not, the animation process continues: the program calculates new ball positions, the screen page changes and the pool balls move across the screen.

POOL.ASM — *The* EachBall *Function*

The *EachBall* function displays each one of the pool balls on the screen. First it must clear the old pool ball from the video page that is not being displayed. An important thing to understand here is that, if you are using the page flipping technique, the changing of the screen data always occurs in the part of video memory NOT currently being displayed. This way the user never sees the drawing of the object happening on the screen. The function *AdjustBall* is called to clear the pool ball from the display page, then its location on the screen is adjusted to a new location. The AdjustBall function will be discussed in more detail later in the chapter.

After EachBall clears the pool ball from the display page and determines the ball's new location, it translates that location into a new video display address. This calculation entails getting the pool ball's current Y location (vertical location) in CurrentY and offset by the pool ball number in BX. This number is then multiplied by the screen width to determine which scan line the pool ball object is to be displayed on. This number is then added to the current page offset (discussed earlier) to determine the scan line to be updated in video memory. Finally, the X location (horizontal location) in CurrentX is added to it to get the exact location of where to change video memory.

Next, the location of the data representing the pool ball is calculated based upon the ball's width and height, and number. The function DisplayBall is used to display the pool ball in video memory and will be discussed in more detail later in the chapter.

This process is continued until each one of the pool balls has been updated and moved.

POOL.ASM — *The* AdjustBall *Function*

The purpose of this function is to clear the pool ball from the memory page, then calculate the new position of the pool ball. First, the AX register is assigned the previous Y coordinate of the pool ball. This value comes from the variable PreviousY. This value is then multiplied by the screen width to determine the correct scan line to start on. The DI register is assigned to the correct page offset (Page 0 or Page 1) and the value in AX is added to it. Next, the previous X coordinate of the pool ball is retrieved from the variable PreviousX and is also added to the DI register. The video display address in DI is now the location of the pool ball to be cleared.

The SI register is assigned to point at the blank ball image. This image is used to clear the pool ball from video memory. It has the same dimensions (BALL_WIDTH * BALL_HEIGHT) as a pool ball. The pool ball needs to be cleared from video memory because if it is not, then images of the pool ball will be left around in video display memory. The CX and BX registers are assigned the values BALL_WIDTH and BALL_HEIGHT, respectively, and the function DisplayBall is called to display the blank pool ball image to video memory, effectively erasing the ball.

After the pool ball has been erased from the video page, its `CurrentY` and `CurrentX` locations are obtained and assigned to the `PreviousY` and `PreviousX` locations. Next, the function checks the pool ball's repeating variable `NumReps` to see if this ball has moved all it was supposed to move. The `NumReps` variable is the number of times the pool ball is supposed to move with the given X and Y coordinates defined by the `CurrPath` variable array defined in the file `BALLS.INC`. This array will be discussed in more detail later in the chapter.

When the `NumReps` variable reaches `0`, then the movement for the current X and Y directions has finished and a new current path is determined. The `CurrPath` array index is used to check the movement byte for this pool ball. If there are still movement instructions left, then they are used; if not, the `OrigPath` array is used to start at the beginning of the movement path. Next, the `NumReps`, `XIncrement`, `YIncrement` and `CurrPath` arrays are set to contain the correct movement data. Last, the `CurrentX` and `CurrentY` locations are incremented by the `XIncrement` and `YIncrement` variables.

The Include File — `BALLS.INC`

`BALLS.INC` contains the data needed to display each one of the pool balls. It also contains the path (movement) variables associated with each particular ball. This file is very data intensive and for that reason its listing is not completely given in this chapter. For the complete include file please refer to the companion code disk that comes with this book.

The first image defined in the file is the blank ball image. This is nothing more than a dark green square. This image is used to erase individual pool balls from video display memory. The file then defines images for all the balls, beginning with the one ball and continuing until the cue ball. Let's take a look at one of the pool ball images.

I will focus my discussion of the ball definitions on the 15 ball. One of the first things to determine when designing screen images is the size of the object. For the purposes of the pool program, I chose a pool ball 16 pixels wide and 16 pixels high. I chose a 16x16 ball not only because it looks good on the screen, but because 16 is easily divided into two 8-bit bytes. Since there are eight pixels per horizontal display byte, the width of the image needs to be two and the height needs to be 16. Since the program must define all four video planes, each 2 by 16 array is repeated four times, once for each video planes.

According to the rules of pool, the 15 ball is a white ball with a dark red stripe around it and the number 15 in the center of the stripe. Figure 19.1 is the video display image of the 15 ball per plane.

Each one of the data arrays per plane is written to video memory. Before each image is written to video display memory, the VGA port address of `0x3C4` is called to inform the VGA that a write is about to occur on the given color plane. First plane 0 (Blue) is written, then plane 1 (Green), and so on. Experiment with changing these byte patterns to see how they affect the video display of the 15 ball.

The next data definitions in the `BALLS.INC` file are the `CurrentX` and `CurrentY` arrays. These are initialized with the starting locations of each pool ball on the screen. The values in these arrays are changed to keep track of the current X and Y locations of each pool ball. The data definitions of the `PreviousX` and `PreviousY` locations are also initialized

```
|----------------------------------------------------------------------
|           Plane 0             Plane 1             Plane 2            Plane 3
|            Blue                Green                Red             Intensity
|----------------------------------------------------------------------
|                 1111111              1111111              1111111              1111111
|       1234567890123456 1234567890123456 1234567890123456 1234567890123456
|----------------------------------------------------------------------
|  1 | 0000011111100000 1111111111111111 0000011111100000 0000011111100000
|  2 | 0001111111111000 1111111111111111 0001111111111000 0001111111111000
|  3 | 0011111111111100 1111111111111111 0011111111111100 0011111111111100
|  4 | 0000000000000000 1000000000000001 0111111111111110 0000000000000000
|  5 | 0000010011110000 1000010011110001 0111111111111110 0000010011110000
|  6 | 0000110010000000 0000110010000000 1111111111111111 0000110010000000
|  7 | 0000010011100000 0000010011100000 1111111111111111 0000010011100000
|  8 | 0000010000010000 0000010000010000 1111111111111111 0000010000010000
|  9 | 0000010000010000 0000010000010000 1111111111111111 0000010000010000
| 10 | 0000010010010000 0000010010010000 1111111111111111 0000010010010000
| 11 | 0000111001100000 0000111001100000 1111111111111111 0000111001100000
| 12 | 0000000000000000 1000000000000001 0111111111111110 0000000000000000
| 13 | 0000000000000000 1000000000000001 0111111111111110 0000000000000000
| 14 | 0011111111111100 1111111111111111 0011111111111100 0011111111111100
| 15 | 0001111111111000 1111111111111111 0001111111111000 0001111111111000
| 16 | 0000011111100000 1111111111111111 0000011111100000 0000011111100000
```

Figure 19.1 *Video image of the 15 ball in each of the four color planes.*

to the starting locations of each pool ball and are used to keep track of the previous pool ball locations.

The data definitions `XIncrement`, `YIncrement`, and `NumReps` are used to determine the path the pool ball will take. These arrays work in conjunction with the `CurrPath` and `OrigPath` arrays. The `CurrPath` and `OrigPath` arrays point at the data definitions, such as the `FifteenBallPath`, which is used to control the path of each individual pool ball.

The data array `FifteenBallPath` defines the path the 15 ball will take. The data array `CurrentX` has the value 10 and `CurrentY` has the value 182. So, the 15 ball will initially start at (10, 182) on the screen. The `NumReps` array is initialized to `1` and is decremented in the function `AdjustBall`. When this array reaches `0`, the program reads the data array `FifteenBallPath` to get the new `NumReps`, `XIncrement`, and `YIncrement` variables. These values start with 40 for `NumReps`, 0 for the `XIncrement`, and 0 for the `YIncrement`. This means that the pool ball has 40 reps of 0 and 0, which in essence means that the 15 ball will wait 40 turns before advancing to the next movement path of 50, 0, 2. (The wait is so that the cue ball has time to reach the rack.) When `NumReps` reaches 0 again, the new movements are read into the arrays.

When the movement factor read in is initially 0, it means that the pool ball reached the end of its movement array and that we need to start at the beginning again. By adjusting the data in these arrays the programmer can control the path of any given pool ball. Of course, the paths assigned in this example are not very realistic, but by assigning more meaningful numbers to the arrays, you can simulate the look of a real pool game.

`POOL.ASM` — *The* `DisplayBall` *Function*

The `DisplayBall` function is called to actually display a pool ball image to video display memory. All parameters are passed in the registers. The `DS:SI` register pair points at the pool ball image to be displayed. The `ES:DI` register pair points at the video display memory byte at which to begin the display. The `CX` register contains the width of the pool ball and the `BX` register contains the height.

This function will loop through each video display plane (0-3) and put the pool ball image in the plane. The `AX` register is initialized to 1 for the first display plane (BLUE). The height of the pool ball (in `BX`) and the starting video display memory offset (in `DI`) are saved on the stack. These are restored when the next video plane is written. Next the display memory address is saved again along with the video plane in `AX` and the width of the pool ball in `CX`.

These are restored for each new scan line. The `SETREG` macro for the VGA port `0x3C4` is called to enable the video plane for writing. The image at `DS:SI` is moved to `ES:SI` for the width of the pool ball (in `CX`). The `CX`, `AX` and `DI` registers are restored and the `DI` register is advanced to the next scan line by adding the screen width to it. The height of the pool ball (in `BX`) is decremented and the loop is continued until all the image has been moved to the video plane. The original video display memory address (in `DI`) and the height of the pool ball (in `BX`) are restored from the stack, and the `AX` register is multiplied by 2 to advance to the next video display plane. When all of the planes have been written to the pool ball has been displayed.

Conclusion

This concludes the discussion of VGA graphic animation. While this chapter does not even begin to detail all of the abilities of the VGA card, it does show you how to animate objects on the screen. You should now be familiar with the general techniques involved in simple graphic animation. I hope this introduction will entice you to further explore the world of graphic animation.

References

Abrash, Michael. *Power Graphics Programming*, Que Corporation, 1989.

Abrash, Michael. "On Graphics," *Programmer's Journal*, The Resource Journal for IBM Micro Programmers, January through October 1989.

Ladymon, Robert F. "Rick." *Graphic User Interface Programming with C*, Wordware Publishing, June 1992.

Listing 19.1 POOL.ASM

```
;**********************************************************
;*                                                        *
;*  PURPOSE : This file contains the assembly language    *
;*            code needed to perform the pool simulation  *
;*            progam.                                     *
;*                                                        *
;*  COMPILE PROCEDURES : masm pool;                       *
;*                       link pool;                       *
;*                                                        *
;**********************************************************

;**********************************************************
;*                                                        *
;* STACK : The stack segment is set to 512 bytes.         *
;*                                                        *
;**********************************************************

stackseg        segment para stack ’STACK’
          db    512 dup (?)
stackseg        ends

;**********************************************************
;*                                                        *
;*  INCLUDE FILE : EQUATES.INC                            *
;*                                                        *
;*  PURPOSE : This include file is used to define all of  *
;*            the equates needed to definethe VGA port    *
;*            addresses, screen width and height, and     *
;*            the pool ball width and height.             *
;*                                                        *
;**********************************************************

include equates.inc

;**********************************************************
;*                                                        *
;*  INCLUDE FILE : BALLS.INC                              *
;*                                                        *
;*  PURPOSE : This file contains the variable data        *
;*            definitions needed to display and keep      *
;*            track of each pool balls’ location and      *
;*            current path.                               *
;*                                                        *
;**********************************************************
```

Listing 19.1 *continued*

```
include balls.inc

;**********************************************************
;*                                                        *
;*  MACRO: SETREG IR1,IR2                                 *
;*                                                        *
;*  PURPOSE : Macro used to set the VGA port address of   *
;*            the VGA video display card.  It is used     *
;*            to set the indexed register IR1 of the      *
;*            VGA video display chip with the indexed     *
;*            register at IR2 to AL.                      *
;*                                                        *
;**********************************************************

SETREG   macro   IR1,IR2
   mov   dx,IR1               ; Move the indexed register
   mov   ah,al                ; Move AL to AH
   mov   al,IR2               ; Set AL to the map register
   out   dx,ax                ; Perform out at DX with AX
   endm

;**********************************************************
;*                                                        *
;*  Start of the code segment for the pool program        *
;*                                                        *
;**********************************************************

cseg     segment   para public 'CODE'
         assume    cs:cseg, ds:dseg

;**********************************************************
;*                                                        *
;* FUNCTION : Pool                                        *
;*                                                        *
;* PURPOSE  : Main programming loop of the simulated      *
;*            pool game.  Sets the video display mode to  *
;*            VGA mode.  Clears the screen in dark green. *
;*            Performs the animation sequence to simulate *
;*            pool balls moving on the screen.  Checks    *
;*            for a keystroke to exit the program.        *
;*                                                        *
;**********************************************************
```

Listing 19.1 *continued*

```
Pool      proc    near
 mov     ax,dseg              ; Set DS to the data segment
 mov     ds,ax                ; that we are using
 mov     ax,010h              ; Set display to VGA mode
 int     10h                  ; Interrupt to change mode
 mov     ax,0A000h            ; A000h is the display address
 mov     es,ax                ; ES points at video memory
                              ; Clear screen in green for
                              ; background of the pool table
 mov     al,02h               ; Set green plane for changes
 SETREG  VGA_INDEX, WRITING   ; Macro to inform video card
 mov     di,0                 ; DI = start of video memory
 mov     cx,56000             ; CX = # of bytes to clear
 mov     al,0FFh              ; 0FFh = clear all of the bits
 rep     stosb                ; use the repeat function

Animate:                      ; Animate all of the balls
 mov     bx,(NUM_BALLS*2)-2   ; BX = Counter of the balls
 call    EachBall             ; Display each of the balls
 call    WaitForSync          ; Wait for the vertical sync
 mov     [POffset],P0_OFFSET  ; Set page offset to Page 0
 xor     [CurrentPage],1      ; Toggle the current page
 jz      CheckForKey          ; Check for key if Page 0
 mov     [POffset],P1_OFFSET  ; Set page offset to page 1

CheckForKey:
 mov     ah,1                 ; Check for a keystroke
 int     16h                  ; Issue interrupt to check key
 jz      Animate              ; Continue if don't have one
 mov     ah,0                 ; Get the keystroke
 int     16h                  ; Issue interrupt to get key
 mov     ax,3                 ; Set video back text mode
 int     10h                  ; Issue interrupt to set text
 mov     ah,4ch               ; Exit to dos
 int     21h                  ; Pool program has ended

Pool    endp
```

Listing 19.1 *continued*

```
;***********************************************************
;*                                                         *
;* FUNCTION : EachBall                                     *
;*                                                         *
;* PURPOSE  : Displays each one of the pool balls (1 - 15 *
;*            and the cue ball) to the screen.  The        *
;*            function AdjustBall is called to clear the   *
;*            pool ball from its current location and      *
;*            adjust its location to its new location.     *
;*            After adjustment, the pool ball is           *
;*            displayed to the screen.                     *
;*                                                         *
;***********************************************************

EachBall   proc    near

EachBallLoop:
 call    AdjustBall            ; Adjust pool ball location
 mov     ax,[CurrentY + bx]    ; Get the current Y location
 mov     cx,SC_WIDTH           ; Get the screen width
 mul     cx                    ; Multiply to get Y offset
 mov     di,[POffset]          ; Get the current page offset
 add     di,ax                 ; Add the Y offset to it
 mov     ax,[CurrentX + bx]    ; Get the current X offset
 add     di,ax                 ; Add it to the offset
 push    bx                    ; Save current pool ball index
 mov     ax,bx                 ; Calculate which ball image
 mov     bx,2                  ; to display on the screen
 div     bx
 mov     bx,BALL_WIDTH * BALL_HEIGHT * 4
 mul     bx
 mov     si,offset OneBall     ; Get address of first ball
 add     si,ax                 ; Add calculated offset to it
 mov     cx,BALL_WIDTH         ; Get the pool ball width
 mov     bx,BALL_HEIGHT        ; Get the height of the ball
 call    DisplayBall           ; Display the pool ball
 pop     bx                    ; Restore the current ball
 dec     bx                    ; Advance to the next pool
 dec     bx                    ; ball index
 jns     EachBallLoop          ; Do the next ball if not done
 ret

EachBall   endp
```

Listing 19.1 *continued*

```
;**********************************************************
;*                                                        *
;* FUNCTION : AdjustBall                                  *
;*                                                        *
;* PURPOSE  : Adjust the location of the pool ball. Clear *
;*            the pool ball from the display using the    *
;*            blank pool ball image.  Save the current    *
;*            X and Y locations to the previous locations.*
;*            Check to see if it is time to move this     *
;*            pool ball.  Determine the new location of   *
;*            the pool ball and save its location.        *
;*                                                        *
;**********************************************************

AdjustBall      proc   near
 push     bx                    ; Save the pool ball index
 mov      ax,[PreviousY + bx]   ; Get previous Y location
 mov      cx,SC_WIDTH           ; Multiply by screen width for
 mul      cx                    ; correct Y location of ball
 mov      di,[POffset]          ; Get the current page offset
 add      di,ax                 ; Add Y offset to it
 mov      ax,[PreviousX + bx]   ; Get the previous X location
 add      di,ax                 ; Add to get correct location
 mov      si,offset BlankBall   ; Set SI to Blank Ball image
 mov      cx,BALL_WIDTH         ; Get width of pool ball
 mov      bx,BALL_HEIGHT        ; Get height of pool ball
 call     DisplayBall           ; Display the blank ball
 pop      bx                    ; Restore the pool ball index
 mov      ax,[CurrentY + bx]    ; Get the current Y location
 mov      [PreviousY + bx],ax   ; Move to previous Y location
 mov      ax,[CurrentX + bx]    ; Get the current X location
 mov      [PreviousX + bx],ax   ; Move to previous X location
 dec      [NumReps + bx]        ; Check if time to change path
 jnz      EndAdjustBalls        ; If not, then move pool ball
 mov      si,[CurrPath + bx]    ; Get current path for
 lodsw                          ; this pool ball
 and      ax,ax                 ; Check to see if it is zero
 jnz      SetNewMove            ; If not, then use it
 mov      si,[OrigPath + bx]    ; Use the original the path
 lodsw                          ; for this pool ball

SetNewMove:
 mov      [NumReps + bx],ax     ; Save number of repeats left
 lodsw                          ; Get new X increment amount
 mov      [XIncrement+bx],ax    ; Save new X increment amount
 lodsw                          ; Get new Y increment amount
 mov      [YIncrement+bx],ax    ; Save new Y increment amount
 mov      [CurrPath + bx],si    ; Save current ball offset
```

Listing 19.1 *continued*

```
EndAdjustBalls:
 mov     ax,[XIncrement+bx]   ; Get current X increment
 add     [CurrentX + bx],ax   ; Add it to current X location
 mov     ax,[YIncrement+bx]   ; Get current Y increment
 add     [CurrentY + bx],ax   ; Add it to current Y location
 ret

AdjustBall    endp

;***********************************************************
;*                                                         *
;* FUNCTION : DisplayBall                                  *
;*                                                         *
;* ENTRY    : SI = The pool ball image to display          *
;*            DI = Video display address offset            *
;*            CX = Width of the pool ball                  *
;*            BX = Height of the pool ball                 *
;*                                                         *
;* PURPOSE  : To display the given pool ball image on      *
;*            the screen at the given offset.  For each    *
;*            one of the display planes, display the       *
;*            given pool ball image on the video display   *
;*            plane.                                       *
;*                                                         *
;***********************************************************

DisplayBall   proc   near
 mov     ax,01h               ; Initialize to Blue plane

EachPlane:                    ; For each one of the planes
 push    bx                   ; Save the height of the ball
 push    di                   ; Save video address offset

FillPlane:                    ; Fill in bytes for this plane
 push    di                   ; Save video address offset
 push    ax                   ; Save which plane we are on
 push    cx                   ; Save the width of the ball
 SETREG  VGA_INDEX, WRITING   ; Set card for writing bytes
 rep     movsb                ; Move in the width bytes
 pop     cx                   ; Restore the ball width
 pop     ax                   ; Restore the plane we are on
 pop     di                   ; Restore the video address
 add     di,SC_WIDTH          ; Advance to next scan line
 dec     bx                   ; Decrement the ball height
 jnz     FillPlane            ; Do the next video scan line
 pop     di                   ; Restore original address
 pop     bx                   ; Restore the ball height
 shl     ax,1                 ; Advance to next video plane
 cmp     ax,010h              ; Have we done all of them
 jnz     EachPlane            ; Display the next if not
 ret

DisplayBall   endp
```

Listing 19.1 *continued*

```
;**********************************************************
;*                                                        *
;* FUNCTION : WaitForSync                                 *
;*                                                        *
;* PURPOSE  : Waits for the vertical video sync to happen *
;*            then sets the panning wait state to the     *
;*            correct video display address offset.  This *
;*            function is used to unsure a smooth display.*
;*                                                        *
;**********************************************************

WaitForSync proc near
 call   WaitVSync              ; Wait for the vertical sync
 mov    ax,[POffset]           ; Get current page offset
 SETREG SYNC_INDEX, START_LOW  ; Set panning wait state low
 mov    ax,[POffset]           ; Get current page offset
 mov    al,ah
 SETREG SYNC_INDEX,START_HIGH  ; Set wait state for high
 call   WaitVSync              ; Wait for the vertical sync
 ret

WaitForSync endp

;**********************************************************
;*                                                        *
;* FUNCTION : WaitVSync                                   *
;*                                                        *
;* PURPOSE  : Waits for the vertical video sync to happen.*
;*                                                        *
;**********************************************************

WaitVSync   proc   near
 mov     dx,STATUS             ; Get VGA status register

WaitNotVSyncLoop:
 in      al,dx                 ; Check port to see if a
 and     al,VERTICAL           ; vertical sync pulse has come
 jnz     WaitNotVSyncLoop      ; Wait for vertical sync

WaitVSyncLoop:
 in      al,dx                 ; Check port to see if the
 and     al,VERTICAL           ; vertical sync has finished
 jz      WaitVSyncLoop
 ret

WaitVSync  endp
```

Listing 19.1 *continued*

```
;***********************************************************
;*                                                         *
;*  End of the code segment for the pool program           *
;*                                                         *
;***********************************************************

cseg   ends

   end     Pool
```

Listing 19.2 `EQUATES.INC`

```
;**********************************************************
;*                                                        *
;*  PURPOSE : This include file is used to define all of  *
;*            the equates needed to set the VGA port      *
;*            addresses, screen width and height, and     *
;*            the pool ball width and height.             *
;*                                                        *
;**********************************************************

;**********************************************************
;*                                                        *
;* The following equates are used to set the VGA port     *
;* addresses used to communicate with the VGA video       *
;* display card.  A very limited subset is used in the    *
;* simulation pool program.  The equates needed to write  *
;* bytes to the screen and the equates needed to check    *
;* for the vertical sync are defined below.               *
;*                                                        *
;**********************************************************

VGA_INDEX     equ 3c4h      ; VGA write port
WRITING       equ 2         ; Set for write mode

SYNC_INDEX    equ 03d4h     ; Vertical sync index register
START_LOW     equ 0dh       ; Vert. sync address low byte
START_HIGH    equ 0ch       ; Vert. sync address high byte

STATUS        equ 03dah     ; VGA status register
VERTICAL      equ 08h       ; Vertical sync bit

;**********************************************************
;*                                                        *
;* The following equates are used to define the screen    *
;* width and height, the video page offsets, a flag to    *
;* determine which page is being displayed, the pool ball *
;* width and height, and the number of pool balls.        *
;*                                                        *
;**********************************************************

SC_WIDTH      equ 640 / 8   ; Screen width in bytes
SC_HEIGHT     equ 350       ; Screen height in scan lines
P0_OFFSET     equ 0         ; Starting offset for page 0

                            ; Starting offset for Page 1
P1_OFFSET     equ SC_WIDTH * SC_HEIGHT

PAGE1         equ 1         ; Compared to check which page
BALL_WIDTH    equ 2         ; Pool ball width in bytes
BALL_HEIGHT   equ 16        ; Pool ball height
NUM_BALLS     equ 16        ; Number of balls to animate
```

Listing 19.3 BALLS.INC

```
;**********************************************************
;*                                                        *
;*  PURPOSE : This file contains the variable data        *
;*            definitions needed to display and keep      *
;*            track of each pool balls' location and      *
;*            current path.                               *
;*                                                        *
;**********************************************************

dseg segment para common 'DATA'

CurrentPage   db   PAGE1      ; Current page 0 or 1
POffset       dw   P1_OFFSET  ; Initialized to page 1

;**********************************************************
;*                                                        *
;*  The following data definitions define the byte        *
;*  patterns needed to display all of the pool ball       *
;*  objects.  Each object has four planes defined, the    *
;*  Blue plane 0, the Green plane 1, the Red plane 2 and  *
;*  the Intensity plane 3.                                *
;*                                                        *
;**********************************************************

;**********************************************************
; BlankBall - Only the green plane 1 is on
;**********************************************************

BlankBall label byte
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b  ;***************************
   db   00000000b, 00000000b  ; Plane 0 - Blue plane image
   db   00000000b, 00000000b  ;***************************
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
```

Listing 19.3 *continued*

```
;
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b  ;*****************************
   db   11111111b, 11111111b  ; Plane 1 - Green plane image
   db   11111111b, 11111111b  ;*****************************
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
;
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b  ;*****************************
   db   00000000b, 00000000b  ; Plane 2 - Red plane image
   db   00000000b, 00000000b  ;*****************************
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
;
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b  ;*****************************
   db   00000000b, 00000000b  ; Plane 3 - Intensity plane
   db   00000000b, 00000000b  ;*****************************
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
```

Listing 19.3 *continued*

```
;***************************************************************
;
; The version of this file on the code disk includes definitions
; for the One Ball through the Fourteen Ball here.
; In the interest of brevity I have left those definitionsn
; out of this listing. They are similar to the definition of
; the Fifteen Ball (see below). ;
;****************************************************************
; ;
;**********************************************************
; Fifteen ball - Striped dark red with a 15 in the center
;**********************************************************

FifteenBall  label byte
   db   00000111b, 11100000b
   db   00011111b, 11111000b
   db   00111111b, 11111100b
   db   00000000b, 00000000b
   db   00000100b, 11110000b
   db   00001100b, 10000000b
   db   00000100b, 11100000b  ;****************************
   db   00000100b, 00010000b  ; Plane 0 - Blue plane image
   db   00000100b, 00010000b  ;****************************
   db   00000100b, 10010000b
   db   00001110b, 01100000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00111111b, 11111100b
   db   00011111b, 11111000b
   db   00000111b, 11100000b
;
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   10000000b, 00000001b
   db   10000100b, 11110001b
   db   00001100b, 10000000b
   db   00000100b, 11100000b  ;****************************
   db   00000100b, 00010000b  ; Plane 1 - Green plane image
   db   00000100b, 00010000b  ;****************************
   db   00000100b, 10010000b
   db   00001110b, 01100000b
   db   10000000b, 00000001b
   db   10000000b, 00000001b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
```

Listing 19.3 *continued*

```
;
   db   00000111b, 11100000b
   db   00011111b, 11111000b
   db   00111111b, 11111100b
   db   01111111b, 11111110b
   db   01111111b, 11111110b
   db   11111111b, 11111111b
   db   11111111b, 11111111b   ;****************************
   db   11111111b, 11111111b   ; Plane 2 - Red plane image
   db   11111111b, 11111111b   ;****************************
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   01111111b, 11111110b
   db   01111111b, 11111110b
   db   00111111b, 11111100b
   db   00011111b, 11111000b
   db   00000111b, 11100000b
;
   db   00000111b, 11100000b
   db   00011111b, 11111000b
   db   00111111b, 11111100b
   db   00000000b, 00000000b
   db   00000100b, 11110000b
   db   00001100b, 10000000b
   db   00000100b, 11100000b   ;****************************
   db   00000100b, 00010000b   ; Plane 3 - Intensity plane
   db   00000100b, 00010000b   ;****************************
   db   00000100b, 10010000b
   db   00001110b, 01100000b
   db   00000000b, 00000000b
   db   00000000b, 00000000b
   db   00111111b, 11111100b
   db   00011111b, 11111000b
   db   00000111b, 11100000b
```

Listing 19.3 *continued*

```
;***********************************************************
; Cue ball - Solid white
;***********************************************************

CueBall  label byte
   db   00000111b, 11100000b
   db   00011111b, 11111000b
   db   00111111b, 11111100b
   db   01111111b, 11111110b
   db   01111111b, 11111110b
   db   11111111b, 11111111b
   db   11111111b, 11111111b   ;****************************
   db   11111111b, 11111111b   ; Plane 0 - Blue plane image
   db   11111111b, 11111111b   ;****************************
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   01111111b, 11111110b
   db   01111111b, 11111110b
   db   00111111b, 11111100b
   db   00011111b, 11111000b
   db   00000111b, 11100000b
;
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b   ;****************************
   db   11111111b, 11111111b   ; Plane 1 - Green plane image
   db   11111111b, 11111111b   ;****************************
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   11111111b, 11111111b
```

Listing 19.3 *continued*

```
;
   db   00000111b, 11100000b
   db   00011111b, 11111000b
   db   00111111b, 11111100b
   db   01111111b, 11111110b
   db   01111111b, 11111110b
   db   11111111b, 11111111b
   db   11111111b, 11111111b  ;***************************
   db   11111111b, 11111111b  ; Plane 2 - Red plane image
   db   11111111b, 11111111b  ;***************************
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   01111111b, 11111110b
   db   01111111b, 11111110b
   db   00111111b, 11111100b
   db   00011111b, 11111000b
   db   00000111b, 11100000b
;
   db   00000111b, 11100000b
   db   00011111b, 11111000b
   db   00111111b, 11111100b
   db   01111111b, 11111110b
   db   01111111b, 11111110b
   db   11111111b, 11111111b
   db   11111111b, 11111111b  ;***************************
   db   11111111b, 11111111b  ; Plane 3 - Intensity plane
   db   11111111b, 11111111b  ;***************************
   db   11111111b, 11111111b
   db   11111111b, 11111111b
   db   01111111b, 11111110b
   db   01111111b, 11111110b
   db   00111111b, 11111100b
   db   00011111b, 11111000b
   db   00000111b, 11100000b

;**********************************************************
;*                                                        *
;*  The data definitions needed to initialize and keep    *
;*  track of the current X and Y locations are defined by *
;*  the variable arrays CurrentX and CurrentY.  Each one  *
;*  of the pool balls (1 - 15 and the cue ball) is given *
;*  a starting X (horizontal location) and a starting Y   *
;*  (vertical location).                                  *
;*                                                        *
;**********************************************************

CurrentX  dw  18,  16,  16,  14,  14,  14,  12,  12
          dw  12,  12,  10,  10,  10,  10,  10,  60

CurrentY  dw 150, 142, 158, 134, 150, 166, 126, 142
          dw 158, 174, 118, 134, 150, 166, 182, 150
```

Listing 19.3 *continued*

```
;**********************************************************
;*                                                        *
;*  The data definitions needed to initialize and keep    *
;*  track of the previous X and Y locations are defined   *
;*  by the variable arrays PreviousX and PreviousY.  Each *
;*  one of the pool balls (1 - 15 and the cue ball) is    *
;*  given a previous X (horizontal location) and a       *
;*  previous Y (vertical location).  These locations are *
;*  initialized to start at the same place as the         *
;*  CurrentX and CurrentY locations but will be changed   *
;*  to the pool balls' previous location during execution *
;*  of the program.                                       *
;*                                                        *
;**********************************************************

PreviousX  dw  18,  16,  16,  14,  14,  14,  12,  12
           dw  12,  12,  10,  10,  10,  10,  10,  60

PreviousY  dw 150, 142, 158, 134, 150, 166, 126, 142
           dw 158, 174, 118, 134, 150, 166, 182, 150

;**********************************************************
;*                                                        *
;* The data definitions used to keep track of the pool    *
;* balls' current X and Y increments and the number of    *
;* times to repeat the movement are defined by the        *
;* variable arrays XIncrement, YIncrement and NumReps.    *
;*                                                        *
;**********************************************************

XIncrement dw   1,   1,   1,   1,   1,   1,   1,   1
           dw   1,   1,   1,   1,   1,   1,   1,   1

YIncrement dw   1,   1,   1,   1,   1,   1,   1,   1
           dw   1,   1,   1,   1,   1,   1,   1,   1

NumReps    dw   1,   1,   1,   1,   1,   1,   1,   1
           dw   1,   1,   1,   1,   1,   1,   1,   1
```

Listing 19.3 *continued*

```
;**********************************************************
;*                                                        *
;* The data definitions used to keep track of the pool    *
;* balls' current and original paths are defined by the   *
;* variable arrays CurrPath and OrigPath.                 *
;*                                                        *
;**********************************************************

CurrPath   dw  OneBallPath,      TwoBallPath
           dw  ThreeBallPath,    FourBallPath
           dw  FiveBallPath,     SixBallPath
           dw  SevenBallPath,    EightBallPath
           dw  NineBallPath,     TenBallPath
           dw  ElevenBallPath,   TwelveBallPath
           dw  ThirteenBallPath, FourteenBallPath
           dw  FifteenBallPath,  CueBallPath

OrigPath   dw  OneBallPath,      TwoBallPath
           dw  ThreeBallPath,    FourBallPath
           dw  FiveBallPath,     SixBallPath
           dw  SevenBallPath,    EightBallPath
           dw  NineBallPath,     TenBallPath
           dw  ElevenBallPath,   TwelveBallPath
           dw  ThirteenBallPath, FourteenBallPath
           dw  FifteenBallPath,  CueBallPath

;**********************************************************
;*                                                        *
;* The data definitions used to define the individual     *
;* paths of the pool balls are defined for each one of    *
;* pool balls.                                            *
;*                                                        *
;**********************************************************

OneBallPath   label   word
           dw  40,  0,  0, 30,  1, -1, 30,  1,  1
           dw  30, -1,  1, 30, -1, -1,  0

TwoBallPath   label   word
           dw  40,  0,  0, 20,  0, -2, 40,  1, -1
           dw  40, -1,  1, 20,  0,  2,  0

ThreeBallPath  label   word
           dw  40,  0,  0, 20,  0,  2, 40,  1,  1
           dw  40, -1, -1, 20,  0, -2,  0

FourBallPath   label   word
           dw  40,  0,  0, 30,  0, -2, 30,  1, -1
           dw  30, -1,  1, 30,  0,  2,  0

FiveBallPath   label   word
           dw  40,  0,  0, 20,  0,  1, 40,  1, -1
           dw  40, -1,  1, 20,  0, -1,  0
```

Listing 19.3 *continued*

```
SixBallPath   label   word
           dw  40,  0,  0, 30,  0,  2, 30,  1,  1
           dw  30, -1, -1, 30,  0, -2,  0

SevenBallPath   label   word
           dw  40,  0,  0, 40,  0, -2, 20,  1, -1
           dw  20, -1,  1, 40,  0,  2,  0

EightBallPath   label   word
           dw  42,  0,  0, 29,  0, -1, 30,  1, -1
           dw  30, -1,  1, 29,  0,  1,  0

NineBallPath   label   word
           dw  40,  0,  0, 30,  0,  1, 30,  1,  1
           dw  30, -1, -1, 30,  0, -1,  0

TenBallPath   label   word
           dw  40,  0,  0, 40,  0,  2, 20,  1,  1
           dw  20, -1, -1, 40,  0, -2,  0

ElevenBallPath   label   word
           dw  40,  0,  0, 50,  0, -2, 10,  1, -1
           dw  10, -1,  1, 50,  0,  2,  0

TwelveBallPath   label   word
           dw  40,  0,  0, 40,  0, -1, 20,  1, -2
           dw  20, -1,  2, 40,  0,  1,  0

ThirteenBallPath   label   word
           dw  40,  0,  0, 20,  0,  1, 10, -1, -1
           dw  10,  1,  1, 20,  0, -1, 20,  0, -1
           dw  10, -1, -1, 10,  1,  1, 20,  0,  1, 0

FourteenBallPath   label   word
           dw  40,  0,  0, 40,  0,  1, 20,  1,  2
           dw  20, -1, -2, 40,  0, -1,  0

FifteenBallPath   label   word
           dw  40,  0,  0, 50,  0,  2, 10,  1,  1
           dw  10, -1, -1, 50,  0, -2,  0

CueBallPath   label   word
           dw  40, -1,  0, 45,  1,  0, 15, -1,  0
           dw  10,  1,  0, 50,  0,  0,  0

;**********************************************************
;*                                                        *
;* End of data segment for the pool simulation program.  *
;*                                                        *
;**********************************************************

dseg      ends
```

Chapter 20

The Math Coprocessor

Lee Daniel Crocker

This article is for those with some assembly language experience who would like to write code for the floating point coprocessor. The accompanying code was compiled with Microsoft C v6.00 and Microsoft MASM v5.10, though other compilers and assemblers accept the same syntax. The demo program uses a few functions from Microsoft's `GRAPHICS.LIB`. These may be easily modified to any other graphics environment or eliminated altogether without affecting the meat of the issue. This chapter is too brief for an exhaustive treatment of all aspects of floating point math, much less the complete details of the hardware that supports it. It will, I think, give you a leg up on the concepts and show you a little about writing the code. A complete book on the particular processor would give you the details of exactly which bit goes where.

What Is Floating Point?

Computers are fond of whole numbers. A bit is an all-or-nothing affair; on or off. It is a simple matter to use many bits in a pattern like 00010100 to represent a whole number like 20, but how do you represent 3.14?

One way to do it is to treat whole numbers as, say, thousandths. That way, 20 would become 20000 and 3.14 would become 3140. Adding these numbers would produce 23140, which you would interpret as 23.14, the correct answer. This is called "fixed point" math because the number of decimals you are working with is fixed; here it is three. The main advantage of this scheme is that it is fast and simple. The disadvantage is that you need many bytes to store numbers of high accuracy.

"Floating point" is a way to represent numbers in which the number of significant digits is fixed but the actual range is not. For example, six digits could represent the number 123,456,000,000 or the number 0.000123456, but the computer would have to round off 1,234,567 to 1,234,570. Numbers like this are stored in two parts. The first part contains the actual digits (such as 123456), called the "mantissa" (French for "makeweight" — something thrown on the scale with the stuff being weighed to make it balance. Mathematicians love attaching new meanings to common words like that.) The second part contains the position of the decimal point, called the exponent.

Floating point has gained popularity over the years as the preferred method, primarily because of its small size. Arithmetic with these numbers is difficult, so many libraries of routines have been written for dealing with them. These libraries are generally slow and hard to use.

The Math Coprocessor

Fortunately, several microprocessor makers make chips that do most of the work of floating point math for you. Intel does this with the 8087 FPU (Floating Point Unit) series, a set of chips that can be added to your computer to provide these functions. Each of these chips has a set of eight 80-bit registers that hold floating point numbers and a set of instructions for manipulating them. The code in this chapter was developed for the original 8087 chip and is portable to the newer 80287 and 80387 chips. The 80287 and 80387 chips add a few new functions that are very useful for some applications, but a discussion of these enhancements is beyond the scope of this chapter. Intel's 486DX and Pentium chips contain the math functions built in, so adding a coprocessor is unnecessary. The 486SX is a 486 that has had the math functions disabled and therefore needs a coprocessor. The Cyrix company also makes a chip that is a functional replacement for the 80387.

Taking advantage of the 8087 in a high-level language is usually just a matter of telling the compiler to do so. Microsoft C v6.00, for example, will automatically compile code to take advantage of the 8087 if an 8087 is detected at run time. This is ideal for most uses. It allows your program to run on any hardware while still taking advantage of more advanced machines.

A high-level language can produce very good code for the average case, but when you really need every ounce of performance you can wrest from the hardware, you need assembly language. If you are familiar with adding assembly language routines to your programs, you will find that adding 8087 code is really no more difficult and can provide a great speed boost.

A Thoroughly Impractical Example

Listing 20.1 (`NEWT.C`) is a small C program to draw a plot of the domains of attraction of Newton's method applied to the equation $x^n-1=0$. (Fractal nuts who know what that means can skip this paragraph.) Newton's method finds roots of an equation by taking an initial guess, then applying a formula that produces a closer guess. This is repeated until the answer is as close as necessary. The equation $x^n-1=0$ has n roots (in complex numbers). To which root will the method converge? That depends on the initial guess, but not in a simple way. Guesses very near a root will find that one, but farther out, the behavior is less predictable — in fact, it is chaotic. The `NEWT` program uses the coordinates of each point on the screen as an initial guess, then gives every point that converged onto a particular root the same color. This produces a marvelously detailed design.

`NEWT.C` is written with some care. It passes 16-bit pointers to floating point values instead of the 64-bit numbers themselves. The complex multiply, divide, and power routines use efficient algorithms. Still, the program spends much of its time moving floating point numbers around in memory and could be made faster by taking better advantage of the FPU registers.

Listing 20.2 (`FPSUBS.ASM`) does just that. It contains some general-purpose complex math routines for the 8087 (which can be used by themselves) and some assembly language replacements for most of the functions in `NEWT.C`. To run it, uncomment the `#define ASSEMBLY` statement in `NEWT.C`, then assemble `FPSUBS.ASM` and link it with `NEWT`. On my machine (a 20MHz 80386 with Cyrix math chip) it runs about 35 percent faster than the pure C version. That may not seem like tremendous savings, but mathematicians have a habit of, whenever they do something, doing it several thousand times on large piles of data, so the savings pile up. Also, the assembly code here is little more than a translation of the C code. Still more speed could be gained by more finely tuning the assembly language at the expense of some readability and reusability.

Where do the savings come from? Mostly from the use of the FPU's internal registers to pass arguments to and from the complex multiply, divide, and power routines. Memory is slow. Most code — even simple integer code — can be sped up by keeping values in registers and avoiding memory whenever possible.

The FPU Stack

The FPU's eight internal registers are accessed in the novel fashion of a circular stack. A traditional stack like the 8086's grows as large as memory allows and never allows access to items "above" the stack pointer. The FPU stack is more like a lottery wheel with eight positions and a pointer. The spot on the wheel under the pointer is called the stack top, or simply `ST` in the operand field of an FPU instruction. The elements "below" the stack top are accessed as `ST(1)`, `ST(2)`, etc. When a number is pushed onto the stack, the wheel is rotated so that the previous stack top becomes `ST(1)`, and the number is placed into what was `ST(7)`. The previous contents of `ST(7)` are lost. This circular nature can be used to advantage; the `FINCSTP` instruction, for example, can be used to bring the value at `ST(1)` to the stack top for some operation that requires it there and the previous stack top may still be accessed as `ST(7)`. `FDECSTP` will rotate both back to their previous positions.

It is important to keep track of stack usage. The method used in `FPSUBS.ASM` is a Forth-style stack trace. The "--" indicates the stack pointer and the values listed after it are the values of `ST`, `ST(1)`, etc. If the list has fewer than eight items, the others are either "free" or contain values left by the calling program. Any FPU routine must document carefully how many registers it uses so that calling code knows whether it can leave temporaries on the stack.

Like 8086 instructions, FPU instructions can take two, one, or no arguments. The first argument is the "destination" argument and is the one modified by the instruction. Instructions with no argument usually have `ST` as the implied destination. A book on assembly language can give you the exact syntax requirements of each instruction.

The registers themselves can contain a few things besides numbers. For one, the FPU tries to keep track of which registers have been used and marks those that haven't been used "empty". The `FINIT` instruction tags all registers as empty. There are also some special values like infinity and "NaN" (which means "Not a Number") and "indefinite" (which means it's a number, but we don't know which one). Infinities generally behave themselves: infinity or anything is still infinity (even infinity infinity is infinity, not 0). If you try to do math with these other oddities, you will generally not get useful results. NaN does not even equal NaN.

The Status Word

The register used by the FPU to communicate non-numerical information to the host chip is the status word. This internal register contains flags similar to those of the 8086 flags register and must be tested after comparisons and other operations. The first step is to get the status word into an 8086 register. With the 80287 and above, this can be done directly with `FSTSW AX`. With an 8087, the status word must first be written to memory with `FSTSW [memadr]`, then either tested at that memory address or loaded into a register. `FPSUBS.ASM` contains a macro (`@fstat`) that does this and does a `SAHF` instruction.

The `SAHF` instruction stores the contents of the `AH` register into the flags register. `FPSUBS.ASM` uses this to copy the high-order byte of the FPU status word into the 8086 flags register. The bits in the FPU status word that need to be tested after a comparison happen to line up with the Zero, Carry, and Parity bits of the 8086 flags register in a convenient way. After a comparison, a `JE` instruction will jump if the two operands are equal, `JA` ("jump if above") will jump if the destination is greater than the source, and `JB` ("jump if below") will jump if the destination is less than the source. It is important to note that although `JA`, `JB`, and so on are for unsigned integer comparisons on the 8086, all comparisons on the FPU are signed. The unsigned jump instructions are used only because of the way the flags happen to line up after the `SAHF` operation.

Other bits in the status word can be tested by loading the whole word into a register and testing the bits explicitly. For example, a more detailed examination of the status word than the simple `SAHF` trick above will tell you if the two operands of a compare instruction are not comparable (say one of them was an empty register or NaN). The `FXAM` instruction will look at one number on the top of the stack and set bits based on its validity, sign, and other attributes. `FPSUBS.ASM` contains a function `exam()`, which takes a pointer to a floating point number and returns a number from 0 to 15 based on the `FXAM` instruction. See Figure 20.1.

The low-order byte of the status word contains exception flags. These are described later.

Memory Synchronization

The CPU and FPU can operate in parallel to a small extent. Most of the time that fact is invisible to the programmer, but there is one case that requires attention. When a coprocessor instruction writes to memory, that memory cannot be accessed immediately by a non-coprocessor instruction. The `FWAIT` instruction must be used to wait until the coprocessor completes its write. You will see `FWAIT` used in `FPSUBS.ASM` at the end of the C-callable routines that write their return values to memory. The `FSTSW [memadr]` instruction is a special case. Because it is almost always followed by a non-coprocessor read of the address just written to, it does the synchronization automatically. That is why there is no `FWAIT` instruction in the `@fstat` macro. Memory synchronization is not needed by the 80287 or any later chips and will therefore be unnecessary for most of you.

```
exam()           Status of number
------           ----------------

0                Valid, positive, unnormalized (may
                 cause underflow)
2                Valid, negative, unnormalized
1, 3, 12, 14     Invalid
4                Valid, positive
5                Positive infinity
6                Valid, negative
7                Negative infinity
8, 10            Zero
9, 11, 13, 15    Empty register
```

Figure 20.1 ***Floating point status values.***

Detection and Emulation

Making a program work the same way whether an FPU is present or not can be done in many ways. The standard libraries from Microsoft C compile interrupt instructions that call routines to simulate the behavior of the coprocessor. If these routines detect a genuine coprocessor (using code such as that in the function `fpu()` from `FPSUBS.ASM`), they replace the interrupt calls with real FPU instructions that will be called next time through the same code.

On the 80286 and later processors, there is a much more efficient way. These processors generate a specific interrupt each time an FPU instruction is encountered that can be used to call a general-purpose emulator installed by the user (or more typically, by the operating system). The advantage of this is that the compiler can always generate direct FPU instructions that will run at full speed on the first pass. (Some operating systems have such emulators built in so that code written for them can always assume the presence of an FPU.)

One could use a routine such as `fpu()` to detect an FPU at the start and then run one of two completely separate programs. Microsoft libraries contain an "Alternate" math package that provides extra speed for programs that know they are running without an FPU. Note that `fpu()` will return a positive result on Windows v3.1 or other systems with a built-in emulator regardless of the actual hardware.

The Control Word

You can set some operating parameters of the FPU by writing to the control word. The control word determines things like rounding mode, type of infinities, precision, and exception masks (see the following section). In `FPSUBS.ASM` are two simple routines to get and set the control word. If, for example, you wanted to set the FPU into truncate-toward-zero mode instead of the default even/odd statistician's rounding method, you would call `setcw(getcw() | 0x0600)`.

The low-order byte of the control word contains bits that set exception masks, in the same positions as the exception flags in the status word. The `setcw()` function can be used on these as well, although it is not recommended because it will interfere with the run time library's standard exception handlers.

Exceptions

Many programmers are used to treating calculations as bulletproof because integers are very forgiving, often generating correct or acceptable results even when limits overflow. A lot more can go wrong with floating point numbers than with fixed point ones. The FPU can generate six types of errors. See Figure 20.2.

There are six types of exceptions, or error conditions, that can occur in FPU code. The FPU defaults to having all of these masked. When an exception occurs, if the corresponding bit in the control word is set, the FPU simply takes what it considers to be a reasonable default action (such as returning NaN or infinity or ignoring precision loss) and does not interrupt the execution of the program. If the bit is not set, the FPU generates an interrupt and gives the CPU a chance to deal with the error.

Even if exceptions are masked, exception flags in the status word will be set so you can detect errors by clearing all exceptions, masking all exceptions, performing a calculation, then examining the flags. The exception flags in the status word are cumulative, so they will reflect all errors that have occurred since the last `FCLEX` instruction (or `FINIT`, `FRSTOR`, etc.). Be sure that any routines you are calling don't clear exception flags (which is unlikely) by doing an `FINIT` or similar instruction.

An example of how to detect exceptions in this way is provided by the function `square()` in `FPSUBS.ASM`. Notice that squaring `DBL_MAX` on some machines may only produce a precision error rather than a real overflow. This is because the internal format of the FPU is long double. If you put `LDBL_MAX` into the sample, you would get true infinities.

Invalid Operation	(One of the operands was NaN or empty, etc.)
Denormalized Operand	(Some precision is lost)
Zero Divide	(Good old divide-by-zero)
Overflow	(Result is too big to represent)
Underflow	(Result is too small to represent)
Precision	(True rounding error has occurred)

Figure 20.2 *The six FPU error types.*

In Closing

The particulars of each coprocessor instruction can be found in many books on assembly language, including the *Microsoft Macro Assembler User's Manual* and the quick reference that comes with it. I hope I have passed on a few insights and a few tricks here that you may find useful. Listing 20.2 should tie up any loose ends.

Listing 20.1 newt.c

```
/**********
** Lee Daniel Crocker
*/

#include <stdlib.h>
#include <conio.h>
#include <graph.h>
#include <float.h>
#include <math.h>

typedef struct { double r, i; } COMPLEX;
typedef double REAL;

#define ASSEMBLY 1

#define DEGREE  8
#define ILIMIT  1000    /* Iteration count limit.   */
#define EPSILON 0.000001

#define PI      3.14159265358979323846

#define WIDTH   320     /* Assume CGA.              */
#define HEIGHT  200
#define MODE    _MRES4COLOR

#define YMIN    -3.0    /* Rectangular section      */
#define YMAX    3.0     /* of complex plane.        */
#define XMIN    -4.0
#define XMAX    4.0

COMPLEX roots[DEGREE];  /* Precalculated values.    */
REAL eps2;

#ifdef ASSEMBLY

extern void cmult(COMPLEX *, COMPLEX *, COMPLEX *);
extern int cdiv(COMPLEX *, COMPLEX *, COMPLEX *);
extern void cpower(COMPLEX *, unsigned, COMPLEX *);
extern int calcnewton(COMPLEX *);
extern int fpu(void);
extern int getcw(void);
extern void setcw(short);
extern int exam(REAL *);
extern int square(REAL *, REAL *);

#else
```

Listing 20.1 *continued*

```
/* In all of the complex math procedures below, complex arguments
** are passed through pointers for efficiency. Any or all of the
** arguments passed may point to the same structure without
** affecting the result, and only the structure pointed to by
** the result argument is modified (unless, of course, the
** same pointer is also a non-result argument).
*/
/* Multiply two complex numbers producing a complex
** result.  Knuth's trick of doing it with three
** multiplies and seven adds doesn't help here because
** the 87's multiply is not much slower that its add.
*/

void cmult(COMPLEX *z1, COMPLEX *z2, COMPLEX *result)
{
    REAL tr;

    tr = z1->r * z2->r - z1->i * z2->i;
    result->i = z1->r * z2->i + z1->i * z2->r;
    result->r = tr;
}

/* Divide two complex numbers producing a complex result.
** returns 0 if all is well, -1 if overflow would occur.
*/

int cdiv(COMPLEX *z1, COMPLEX *z2, COMPLEX *result)
{
    COMPLEX tc;
    REAL m, tr;

    m = (z2->r * z2->r) + (z2->i * z2->i);
    if (m < FLT_MIN) return -1;
    else m = 1.0 / m;

    tc = *z2;
    tr = m * (z1->r * tc.r + z1->i * tc.i);
    result->i = m * (z1->i * tc.r - z1->r * tc.i);
    result->r = tr;

    return 0;
}
```

Listing 20.1 *continued*

```
/* Raise complex number to integer power.  Operates by repeatedly
** squaring the base, multiplying that by the result for every bit
** set in the exponent, therefore running in O(log2(exp)) time.
*/

void cpower(COMPLEX *base, unsigned exp, COMPLEX *result)
{
    REAL xt, yt, t2;

    xt = base->r;   yt = base->i;

    if (exp & 1) {
        result->r = xt;
        result->i = yt;
    } else {
        result->r = 1.0;
        result->i = 0.0;
    }
    exp >>= 1;

    while (exp) {
        t2 = (xt + yt) * (xt - yt);
        yt = 2 * xt * yt;
        xt = t2;

        if (exp & 1) {
            t2 = xt * result->r - yt * result->i;
            result->i = result->i * xt + yt * result->r;
            result->r = t2;
        }
        exp >>= 1;
    }
}
```

Listing 20.1 *continued*

```
/* Calculate one iteration.  The formula is z(n+1)= z(n) - f(z(n)) / f'(z(n),
** where z(n)=old, z(n+1)=new, f(z) = z^DEGREE, and f'(z) is the first
** derivative of f(z), in this case DEGREE*z^(DEGREE-1). Both old and new
** are modified, and must point to different structures. Returns 1 if new
** point is sufficently close to old, or if a divide overflow occurred.
*/

int onenewton(COMPLEX *old, COMPLEX *new)
{
    COMPLEX tc;             /* Temporary complex */

    /* Set new=old^DEGREE, tc=old^(DEGREE-1).
    */
    cpower(old, DEGREE-1, &tc);
    cmult(&tc, old, new);

    new->r = new->r * (DEGREE-1) + 1.0;
    new->i *= (DEGREE-1);

    tc.r *= DEGREE;
    tc.i *= DEGREE;

    cdiv(new, &tc, new);

    if (fabs(new->r - old->r) < EPSILON &&
        fabs(new->i - old->i) < EPSILON) return 1;
    else return 0;
}

/* Calculate the color of the point given. Argument is not modified.
*/

int calcnewton(COMPLEX *point)
{
    int i, root, iterations;
    COMPLEX old, new;

    old = *point;
    iterations = 0;

    while (++iterations < ILIMIT) {
        if (onenewton(&old, &new)) break;
        else old = new;
    }
```

Listing 20.1 *continued*

```
    /* To which root did we converge? Begin by assuming root 0,
    ** then test each of the other roots exitingm if one of them
    ** works. Thus, we don't ever have to test root 0 explicitly.
    */
    root = 0;
    for (i=DEGREE-1; i>0; --i) {
        if (fabs(new.r - roots[i].r) < eps2 &&
            fabs(new.i - roots[i].i) < eps2) {
            root = i;
            break;
        }
    }

    /* At this point, <iterations> has the iteration count and <root> has
    ** the root number. These can be combined in various ways to produce
    ** a color for interesting effects. In this case, we will simply
    ** assign a color based on the root, ignoring the iteration count.
    */
    return root;
}

#endif  /* ASSEMBLY */

/* Main program:
*/

int newtonplot(void)
{
    int i, px, py, color;
    COMPLEX point;
    REAL xinc, yinc, theta, dtheta;

    point.r = XMIN;
    point.i = YMIN;

    /* X and Y increments */

    xinc = (XMAX - XMIN) / (WIDTH - 1);
    yinc = (YMAX - YMIN) / (HEIGHT - 1);

    eps2 = 0.5 / DEGREE;
    theta = dtheta = (2.0 * PI) / DEGREE;
    roots[0].r = 1.0; roots[0].i = 0.0;

    for (i = DEGREE-1; i > 0; --i) {
        roots[i].r = cos(theta);
        roots[i].i = sin(theta);
        theta += dtheta;
    }
```

Listing 20.1 *continued*

```
    _setvideomode(MODE);

    for (py = 0; py < HEIGHT; ++py) {
        for (px = 0; px < WIDTH; ++px) {
            color = calcnewton(&point);

            _setcolor(color);
            _setpixel(px, py);

            point.r += xinc;

            if (kbhit()) break;
        }
        if (kbhit()) break;

        point.i += yinc;
        point.r = XMIN;
    }

    getch();
    _setvideomode(_DEFAULTMODE);

    return 0;
}

int main(int argc, char *argv[])
{
    int r;
    double d1, d2, d3;

    printf("FPU: %d\n", fpu());

    d1 = -4.0;
    r = square(&d1, &d3);
    printf("%04x %g\n", r, d3);     /* OK */

    d2 = DBL_MAX;
    r = square(&d2, &d3);
    printf("%04x %g\n", r, d3);     /* Overflow */

    printf("%g, %g, %g\n", d1, d2, d3);
    printf("%d, %d, %d\n", exam(&d1), exam(&d2), exam(&d2));

    r = square(&d3, &d1);
    printf("%04x %g\n", r, d3);     /* Invalid */

    getch();
    return newtonplot();
}

/* End of File */
```

Listing 20.2 fpsubs.asm

```
;**********
; Lee Daniel Crocker
;
; Tabs: 6
;

.model small, c
.8087

@fstat macro
      fstsw [status]          ; Can be replaced with
      mov   ax, [status]      ; fstsw ax on 287
      sahf
      endm

bptr  equ   <byte ptr>
wptr  equ   <word ptr>
qptr  equ   <qword ptr> ; To save typing

.data

degree      dw    8
degreem1    dw    7
ilimit      dw    1000
status      dw    0

floatmin    dq    1.175494351e-38
epsilon     dq    0.000001

oldr        dq    ?
oldi        dq    ?
newr        dq    ?
newi        dq    ?

extrn       eps2:qword, roots:qword

;**********
;
; C-callable functions. Pretty straightforward stuff:
; just load the parameters onto the 8087 stack, call the routine
; that does the work, and pop the result back to memory.
;

.code
```

Listing 20.2 *continued*

```
;
; void cmult(COMPLEX *f1, COMPLEX *f2, COMPLEX *result);
;

cmult proc f1:ptr qword, f2:ptr qword, result:ptr qword
      mov   bx, [f1]
      fld   qptr [bx+8]
      fld   qptr [bx]

      mov   bx, [f2]
      fld   qptr [bx+8]
      fld   qptr [bx]

      call  docmult

      mov   bx, [result]
      fstp  qptr [bx]
      fstp  qptr [bx+8]
      fwait             ; synchronize memory write
      ret
cmult endp

;
; int cdiv(COMPLEX *f1, COMPLEX *f2, COMPLEX *result);
;

cdiv proc f1:ptr qword, f2:ptr qword, result:ptr qword
      mov   bx, [f1]
      fld   qptr [bx+8]
      fld   qptr [bx]

      mov   bx, [f2]
      fld   qptr [bx+8]
      fld   qptr [bx]

      call  docdiv

      mov   bx, [result]
      fstp  qptr [bx]
      fstp  qptr [bx+8]
      fwait             ; synchronize memory write
      ret
cdiv endp
```

Listing 20.2 *continued*

```
;
; void cpower(COMPLEX *base, unsigned exp, COMPLEX *result);
;

cpower proc base:ptr qword, exp:word, result:ptr qword
      mov   bx, [base]
      fld   qptr [bx+8]
      fld   qptr [bx]

      mov   ax, [exp]
      call  docpower

      mov   bx, [result]
      fstp  qptr [bx]
      fstp  qptr [bx+8]
      fwait              ; synchronize memory write
      ret
cpower endp

;
;
;

onenewton:
      fld   [oldi]
      fld   [oldr]
      mov   ax, [degreem1]
      call  docpower

      fld   st(1)
      fld   st(1)
      fld   [oldi]
      fld   [oldr]
      call  docmult     ; - nr  ni  tr  ti

      fild  [degreem1]
      fmul  st(2), st
      fmul
      fld1
      fadd

      fild  [degree]
      fmul  st(4), st
      fmulp st(3), st

      fincstp
      fxch  st(2)
      fdecstp
      fxch  st(2)
      call  docdiv
```

Listing 20.2 *continued*

```
        sub   dx, dx
        or    ax, ax       ; cdiv's return
        jnz   overf

        fld   [oldr]
        fsub  st, st(1)
        fabs
        fcomp [epsilon]
        @fstat
        ja    onout

        fld   [oldi]
        fsub  st, st(2)
        fabs
        fcomp [epsilon]
        @fstat
        ja    onout
overf:
        inc   dx
onout:
        ret

;
; int calcnewton(COMPLEX *point);
;

calcnewton proc point:ptr qword
        mov   bx, [point]
        fld   qptr [bx]
        fstp  [oldr]
        fld   qptr [bx+8]
        fstp  [oldi]

        mov   cx, [ilimit]
ltop:
        call  onenewton

        or    dx, dx
        jz    lbot

        fstp  [newr]
        fstp  [newi]
        jmp   short lbreak
lbot:
        fst   [newr]
        fstp  [oldr]
        fst   [newi]
        fstp  [oldi]
        loop  ltop
```

Listing 20.2 *continued*

```
lbreak:
        mov   dx, [ilimit]
        sub   dx, cx

        mov   bx, [degreeml]
        mov   cl, 4
        shl   bx, cl
        mov   cx, [degreeml]
        add   bx, offset DGROUP:roots
        fld   [eps2]
        fld   [newi]
        fld   [newr]
l2top:
        fld   qptr [bx]
        fsub  st, st(1)
        fabs
        fcomp st(3)
        @fstat
        ja    l2next

        fld   qptr [bx+8]
        fsub  st, st(2)
        fabs
        fcomp st(3)
        @fstat
        jna   l2out
l2next:
        sub   bx, 16
        loop  l2top
l2out:
        ffree st(2)
        ffree st(1)
        ffree st    ; Now dx=iterations and cx=root.
        mov   ax, cx
        and   ax, 3
        ret
calcnewton endp

;
; Subroutines used in various functions:
;
; Multiply complex [a,b] by [c,d]. Uses only two extra stack elements.
;
                        ; - FORTH-style 8087
                        ; - stack trace:
```

Listing 20.2 *continued*

```
docmult:                ; - a  b  c  d
        fld   st(2)     ; - c  a  b  c  d
        fmul  st, st(1) ; - ac  a  b  c  d
        fld   st(4)     ; - d  ac  a  b  c  d
        fmul  st, st(3) ; - bd  ac  a  b  c  d
        fsub            ; - rx=(ac-bd)  a  b  c  d
        fincstp         ; - a  b  c  d  ?  ?  ?  rx
        fmulp st(3), st ; - b  c  ad  ?  ?  ?  rx
        fmul            ; - bc  ad  ?  ?  ?  rx
        fadd            ; - ry=(ad+bc)  ?  ?  ?  rx
        fld   st(4)     ; - rx  ry  ?  ?  ?  rx
        ffree st(5)     ; - rx  ry
        ret

;
; Divide complex [c,d] by [a,b]. Uses every stack element.
;

docdiv:                 ; - a  b  c  d
        fld   st(1)     ; - b  a  b  c  d
        fmul  st, st(2) ; - bb  a  b  c  d
        fld   st(1)     ; - a  bb  a  b  c  d
        fmul  st, st(2) ; - aa  bb  a  b  c  d
        fadd            ; - m  a  b  c  d

        fcom  [floatmin] ; Note: m is always
        @fstat          ; positive here.
        jb    overflow

        fld1            ; - 1  m  a  b  c  d
        fdivr           ; - m  a  b  c  d
        fstp  st(5)     ; - a  b  c  d  m
        fincstp         ; - b  c  d  m  ?  ?  ?  a
        fchs            ; - -b  c  d  m  ?  ?  ?  a
        fdecstp         ; - a  -b  c  d  m
        call  docmult   ; - rx  ry  m
        fxch  st(2)     ; - m  ry  rx
        fmul  st(1), st ; - m  m*ry  rx
        fmul  st, st(2) ; - m*rx  m*ry  rx
        ffree st(2)     ; - rx  ry

        sub   ax, ax
        ret
overflow:
        sbb   ax, ax    ; sets AX=-1
        ret
```

Listing 20.2 *continued*

```
;
; Raise complex [a,b] to integer power in AX. Uses every stack element.
;

docpower:               ; - a  b

        shr   ax, 1 ; Shift bit 0 of exp into CF.
        jc    bit0

        fldz            ; - ry=0  a  b
        fld1            ; - rx=1  ry=0  a  b
        jmp   short whiletop
bit0:
        fld   st(1)     ; - ry=b  a  b
        fld   st(1)     ; - rx=a  ry=b  a  b

whiletop:           ; Square [a,b].

        fld   st(2)     ; - a  rx  ry  a  b
        fadd  st, st(4) ; - (a+b)  rx  ry  a  b
        fld   st(3)     ; - a  (a+b)  rx  ry  a  b
        fsub  st, st(5) ; - (a-b)  (a+b)  rx  ry  a  b
        fmul            ; - t2  rx  ry  a  b
        fxch  st(3)     ; - a  rx  ry  t2  b
        fmul  st, st(4) ; - ab  rx  ry  t2  b
        fst   st(4)     ; - ab  rx  ry  t2  ab
        faddp st(4), st ; - rx  ry  t2=a  2ab=b

        shr   ax, 1     ; Shift next bit of exp into CF
        jnc   nomult    ; If bit is set, multiply result
                        ; by [a,b].

        fld   st(2)     ; - a  rx  ry  a  b
        fmul  st, st(1) ; - a*rx  rx  ry  a  b
        fld   st(4)     ; - b  a*rx  rx  ry  a  b
        fmul  st, st(3) ; - b*ry  a*rx  rx  ry  a  b
        fsub            ; - t2  rx  ry  a  b
        fld   st(3)     ; - a  t2  rx  ry  a  b
        fmulp st(3), st ; - t2  rx  a*ry  a  b

        fld   st(4)     ; - b  t2  rx  a*ry  a  b
        fmul  st, st(2) ; - b*rx  t2  rx  a*ry  a  b
        faddp st(3), st ; - t2  rx  ry  a  b

        fstp  st(1)     ; - rx  ry  a  b
nomult:
        or    ax, ax    ; Set ZF if no more exp.
        jnz   whiletop
```

Listing 20.2 *continued*

```
      ffree st(3)       ; - rx  ry  a
      ffree st(2)       ; - rx  ry
      ret

;
; int fpu(void);  Returns 1 if FPU is present.
;

fpu   proc
      push  si

      fninit
      mov   si, offset status
      mov   wptr [si], 5a5ah  ; Any non-zero value
      fnstsw [si]
      cmp   bptr [si], 0      ; Correctly zeroed result?
      jne   nofpu

      fnstcw [si]
      mov   ax, [si]
      and   ax, 103fh         ; Mask known bits
      cmp   ax, 3fh
      jne   nofpu

      mov   ax, 1             ; Got one!
      jmp   fpdone
nofpu:
      sub   ax, ax
fpdone:
      pop   si
      ret
fpu   endp

;
; int getcw(void);
;

getcw proc
      fnstcw [status]         ; Get FPU control word
      mov   ax, [status]
      ret
getcw endp

;
; void setcw(bits);
;

setcw proc bits:word
      mov   ax, [bits]
      fldcw [status]          ; Set FPU control word
      ret
setcw endp
```

Listing 20.2 *continued*

```
;
; int exam(double *f1);
;

exam  proc f1:ptr qword
      mov   bx, [f1]
      fld   qptr [bx]
      fxam                    ; Set condition codes

      fstsw [status]          ; Can be replaced with
      mov   ax, [status]      ; fstsw ax on 287

      and   ax, 0700h         ; Shift bits around to get
      mov   bptr [status], ah ; a number from 0..15
      mov   al, bptr [status+1]
      shr   al, 1
      shr   al, 1
      shr   al, 1
      and   al, 08h
      or    al, bptr [status]
      sub   ah, ah

      fstp  st(1)             ; Clean stack
      ret
exam  endp

;
; int square(REAL *f1, REAL *result);
;

square proc f1:ptr qword, result:ptr qword
      mov   bx, [f1]
      fld   qptr [bx]

      fnclex                  ; Clear and mask exceptions
      fnstcw [status]
      or    [status], 3fh
      fldcw [status]

      fmul  st, st

      fnstsw [status]         ; Get flags
      mov   ax, [status]
      and   ax, 3fh

      mov   bx, [result]
      fstp  qptr [bx]
      fwait                   ; synchronize memory write
      ret
square endp

end
```

Chapter 21

Interfacing to the PC Floppy Disk Controller

Larry Widing

Although you may use software interrupts to perform disk I/O on the PC, the I/O itself is not interrupt-driven; it is synchronous. In other words, your program can't do anything else while the disk is reading or writing data. The hardware (disk and DMA controller) itself operates asynchronously, but the PC BIOS interface waits for each I/O to complete.

If you are willing to bypass the BIOS and program the disk controller directly, you can create an interrupt-driven interface. That is how programs like FastBack and Central Point Backup achieve their speed; while the floppy drive is writing the first track of backup data, they are preparing the data for the next track to be written. I discovered techniques for implementing asynchronous I/O while developing a program to compete with other high-speed backup programs.

This chapter shows how to program the PC Floppy Disk Controller (FDC). A couple of simplifications keep the code short. First, the code only covers 360Kb drives; supporting other drive types just requires changing a few of the parameters passed to the controller. Second, since a true asynchronous I/O interface can be complex (involving semaphores, events, threads, and so on), this article just presents a barebones interface; you must tailor the code to your particular application. I developed the code in this chapter using Borland's Turbo C++ v1.0 and Turbo Assembler v2.0, but it should port easily to most C compilers.

The Floppy Disk Controller

The Floppy Disk Controller in the IBM PC and clones uses either the NEC PD765 FDC chip or the Intel 8272 controller chip. The FDC supports up to four drives. When I refer to the FDC, I mean both the actual FDC chip (Intel or NEC) and also the support circuitry that controls the drive motors and the DMA and PC interrupt requests.

You communicate with the FDC via three 1-byte I/O ports:

`0x3f2`	`FDC digital output register`
`0x3f4`	`FDC main status register`
`0x3f5`	`FDC data register`

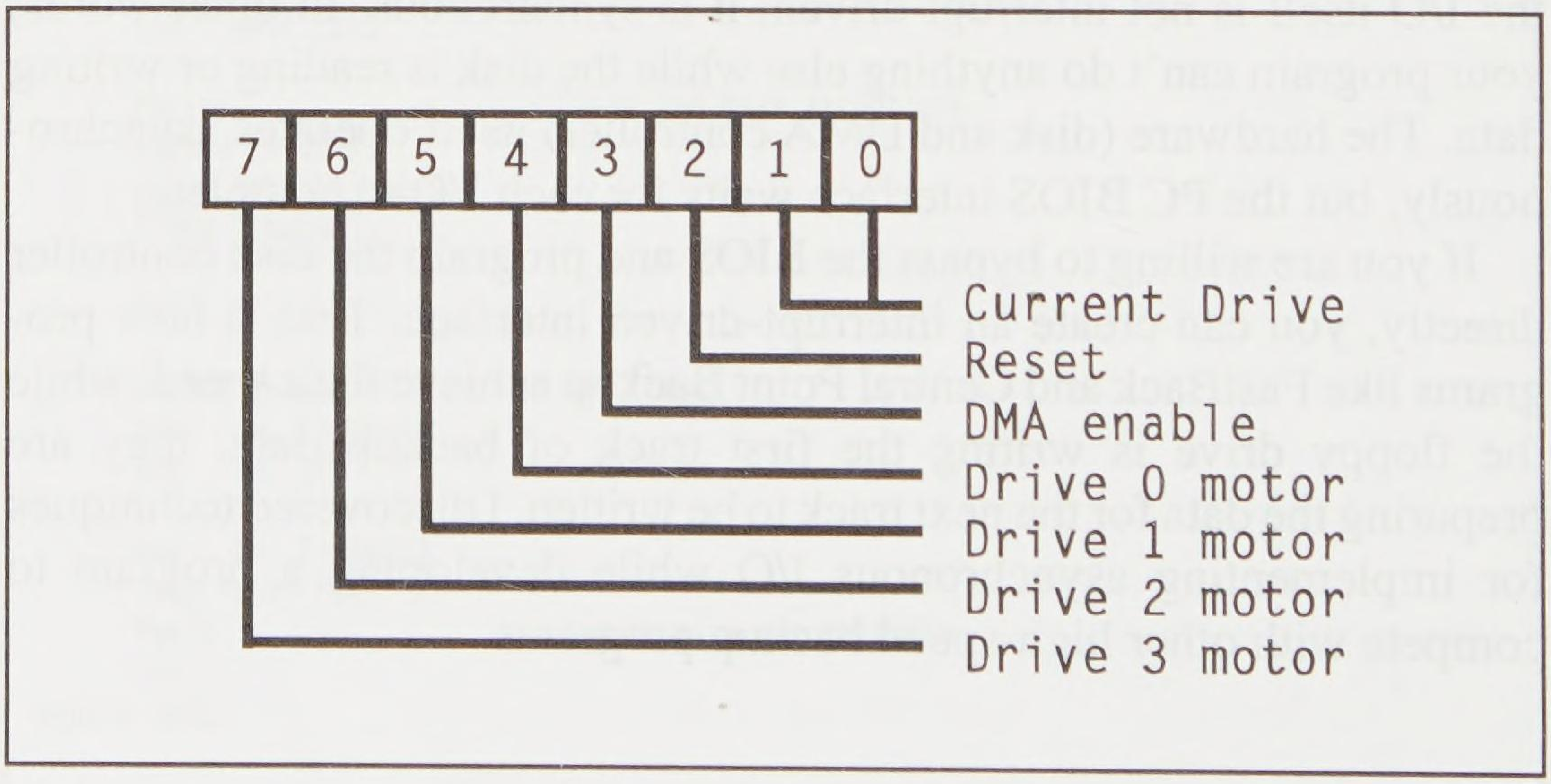


Figure 21.1 *The FDC Digital Ouput Register.*

The Digital Output Register controls some parts of the drive that the controller chip doesn't. For example, four of the bits in the digital output register directly control whether the motor in the corresponding floppy drive is on or off. The FDC main status register constantly reflects aspects of the FDC status, such as whether I/O is in progress. You use the FDC data register to send commands to the FDC and to retrieve the status of completed commands. A more detailed look at these registers follows.

The FDC Digital Output Register (`DOR`) is a write-only 1-byte control register, shown in Figure 21.1. The high four bits each control the motor for one of the four potential floppy drives. Set the bit to turn on the corresponding motor and clear the bit to turn it off. Bit `3` enables the FDC interrupt and the DMA connection to the FDC. You will usually keep this bit on. Clearing bit `2` forces the FDC into a reset state until you set the bit again. Finally, bits `0` and `1` together specify the current drive (`0` through `3`) for FDC commands.

When you write to the `DOR`, you must keep in mind what bits you have already set or cleared. For example, if you want to turn on the motor of drive `0`, you can't just write a `0x10` to the port, since that would also disable the DMA interface, reset the FDC and set the current drive to `0`.

Figure 21.2 shows the Main Status Register (`MSR`), a read-only one-byte register. The lower four bits of the `MSR` show what drives are currently seeking (bit `0` for drive `0`, bit `1` for drive `1`, and so on). The FDC sets bit `4` to `1` when it is processing a read or write request. The FDC sets bit `4` to

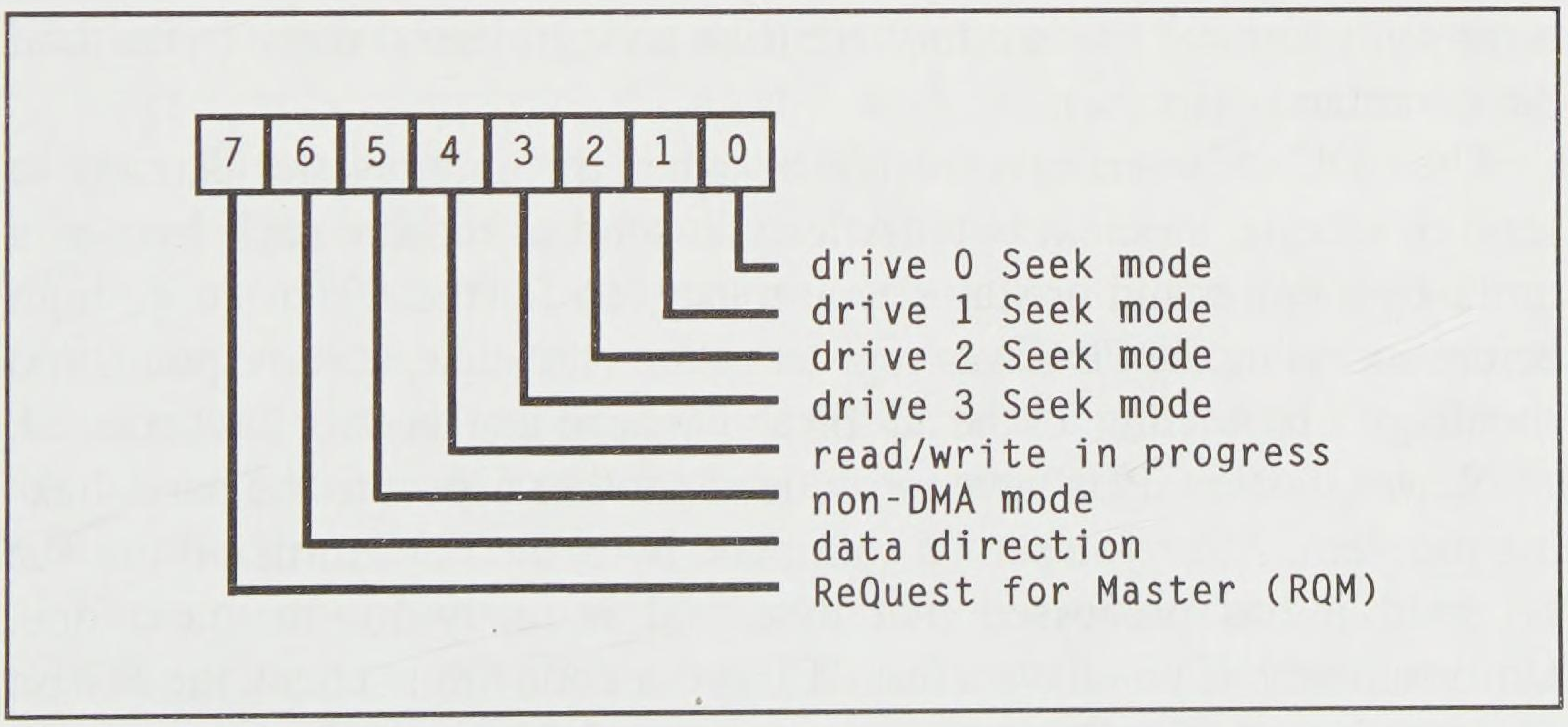


Figure 21.2 ***The FDC Main Status Register.***

0 when it processes data through the DMA channel. Bit 6 indicates whether the FDC data register should be used to send data to the FDC (bit 6 is 0) or receive data from the FDC (bit 6 is 1). The FDC sets bit 7 when the data register is available for reading or writing the next byte.

The FDC data register is a one-byte read/write register. Since most FDC commands contain more than one byte, you must write them to the FDC data register one at a time. Conversely, most FDC commands return more than one byte of status information, so you must read the status bytes from the FDC data register one at a time. When sending a multi-byte command, the FDC stores the bytes in an internal stack until it receives the last command byte, then it executes the command. The FDC uses the same internal stack to hold multi-byte statuses until you retrieve them by reading the data register.

Sending FDC Commands

Sending commands to the FDC requires some consideration of timing. The first timing issue concerns the FDC data register. The CPU can usually send bytes to the FDC faster than the FDC can process them. Whenever two processes such as these are communicating through the same port, they require some form of handshaking.

Two bits in the MSR help with handshaking: the direction bit (bit 6) and the ReQuest for Master (RQM) bit. The direction bit simply shows whether the FDC thinks it should be sending or receiving control information. If you are sending command bytes to the FDC and the data direction bit turns on, you have made a mistake (like trying to send more bytes than the command calls for).

The FDC only turns on the RQM bit when the data register is ready to send or receive the next byte. After you send or receive each byte of a multi-byte command or status, you must wait for the RQM bit to go high before accessing the FDC data register again. Of course, at some point, you should give up waiting for the RQM bit and assume that an error has occurred.

Newer, faster CPUs have introduced another aspect to the handshaking problem. After you send a command byte, the FDC turns off the RQM bit until it has processed that byte and is ready for the next one. Unfortunately, if you have a fast CPU, your code might check the RQM bit again before the FDC has time to turn it off. If that happens, you will send the next command byte before the FDC is ready for it. You need to

Disk Terminology

To program a disk controller, you must understand the disk mechanism in some detail. Some basic terms used to describe the different parts of a diskette are:

Track — a circular ring of magnetic material on the surface of a diskette, separated into multiple sectors.

Cylinder — the set of tracks at the same distance from the center of the disk. For double-sided floppy disks, a cylinder has two tracks, one on the top and one on the bottom of the floppy. Since hard disks often have multiple platters, hard disk cylinders may contain more than two tracks.

Sector — each track is divided into equal-sized sectors. Each sector contains an address mark and a data section. The floppy disk controller uses the address mark to locate the desired sector when reading or writing data.

Head — the part of the disk drive that actually reads and writes the data to the diskette. Floppy drives have one head for each side, or track, of the diskette.

Standard Numberings

The DOS disk I/O interface gives you the illusion that all the sectors on a disk are numbered sequentially, starting with sector number zero. Programming the FDC yourself introduces more complications, however. You must turn on the floppy drive motor, tell the FDC to seek the correct cylinder number, program the DMA controller to transfer the correct number of bytes, and issue a read command that tells the FDC which head to use and what address mark to look for on the track.

DOS uses a particular convention for numbering the disk cylinders, heads, and sectors of floppy disks. DOS numbers cylinders from zero, starting with the outermost cylinder. Heads are numbered from zero. Sectors are numbered from one (this is at the physical level, not the logical level provided by DOS I/O services).

These numberings actually reside on the disk itself. The address mark of each sector consists of four bytes that specify the cylinder number, head number, record (or sector) number, and the number of bytes per sector. Except for the sector number, the numbers in the address mark are really just extra verification. You already had to seek to the correct cylinder number before reading or writing, and you must specify either head zero or head one in the read or write command itself. Nevertheless, you still must specify the four-byte address mark in the read or write command, and the command will fail if the FDC does not find an address mark that matches.

Non-Standard Numberings

Although DOS numbers sectors in the fashion just described, you are free to number yours any way you want. When you format an individual track on the floppy, you must pass a table containing the address marks for all the sectors in that track. Even though you must specify head zero or one in the actual FDC command, you can store any head number from zero to 255 in the address mark of a sector. To retrieve the sector, you must specify the same address mark, though.

There are several reasons you might want to use non-standard numberings. Some backup programs store head numbers other than zero to prevent DOS from trying to read their floppies. Almost any non-standard numbering can serve as a form of copy protection. You can format the track for extra sectors (for example, you can squeeze 10 sectors instead of nine onto a 360Kb floppy track). You can set the cylinder number or sector number to large values, like 199.

A slight variation on non-standard numbering is to number the sectors on a track nonsequentially. This technique is known as interleaving and can speed up floppy I/O. For example, if your program retrieves sectors in numerical order but must do some processing between each read, the disk may have already spun past the next sector before you issue the next read. That means you must wait an entire revolution for the next sector to come around to the head again.

Interleaving leaves some time for processing between each sector. For example, a normal sector layout for a 360Kb floppy track would be one, two, three, four, five, six, seven, eight, nine (there are nine tracks per sector on a standard 360Kb floppy). Suppose you use the following sequence instead: one, six, two, seven, three, eight, four, nine, five. With this interleaved sequence, you can do some processing after you read sector one while the head is skipping over sector six, and so on.

insert a delay in the loop that is long enough to avoid this problem, but not so long that it wastes time.

In the early PCs and ATs, this wasn't a problem because both the processor and the bus ran at similar speeds, so if the `RQM` bit was high, you could assume the FDC was ready for the next byte. Today's machines have processors that run much faster than their I/O bus does, and the addition of cached memory makes creating a loop-based delay almost impossible. Either the loop will take too long on a vintage PC, or not be long enough on a more advanced machine.

The code in Listing 21.1 shows what I have found to be the best way to perform this delay. At first glance, this code may seem unusual, but the time needed by the FDC seems to be tied to the speed of the I/O bus, not the processor speed. By reading and writing a byte to the video display, the processor must access memory stored on the I/O bus, which generates the correct delay for all machines. If the display is not a monochrome display, then the code operates on the second video page to prevent snow from appearing on CGA displays. Listing 21.2 shows the C function prototypes for the functions in Listing 21.1. Call `calibrateDelay()` at the start of your program, and then call `fdcDelay()` to generate the desired delay.

The DMA Controller

The DMA controller in the PC transfers data to and from the FDC. You must program the DMA controller before you send a read or write command to the FDC. Because of the design of the DMA controller, it can only transfer, at most, 64Kb of data at a time, and all of that data must fit within a single DMA page. The DMA views memory as a series of non-overlapping 64Kb pages, so address `0000:0000` is DMA page zero, `1000:0000` is DMA page one, and so forth. The function `dmaPageTest()`, in Listing 21.3, determines whether your buffer fits within a single DMA page.

You program the DMA controller via five I/O ports: a mask port, a mode port, an address port, a page port, and a count port. The mask port (`0x0a`) turns DMA channels on and off. Put the channel number (`0` to `3`) in bits `0` and `1`, and then set or clear bit `2` to turn the channel on or off. The FDC uses DMA channel `2` on the PC, though some third-party drives may use a different channel. Use the mode port (`0x0b`) to specify what operation to perform on what channel. The address port (`0x04`) and the page port (`0x81`) point to a memory buffer, specified in terms of a DMA page number and an offset within that page. You pass the count port (`0x05`) one greater than the number of bytes you want to transfer. See `dmaSetup()` in Listing 21.3 for an example of programming the DMA controller with these ports.

Initializing the FDC

The functions in Listing 21.3 provide two different initialization functions for the FDC. The first initialization function is `fdcSetDriveParms()`. You must call this function once before you issue any FDC commands. `fdcSetDriveParms()` saves the old FDC interrupt handler and installs its own. For that reason, your program must call `fdcResetParms()` (which restores the previous interrupt handler) before it terminates, otherwise the interrupt vector will point at code that no longer exists.

`fdcSetDriveParms()` also accepts and stores away three FDC operating parameters: the step rate time (two to 32 milliseconds in 2-ms increments), the head unload time (zero to 480 milliseconds in 32-ms increments), and the head load time (four to 512 milliseconds in 4-ms increments). If you make these values too large, FDC operations will be unnecessarily slow. If you make them too small, you may get errors. A conservative set of values to use is a `12`, `15`, and `2`, respectively.

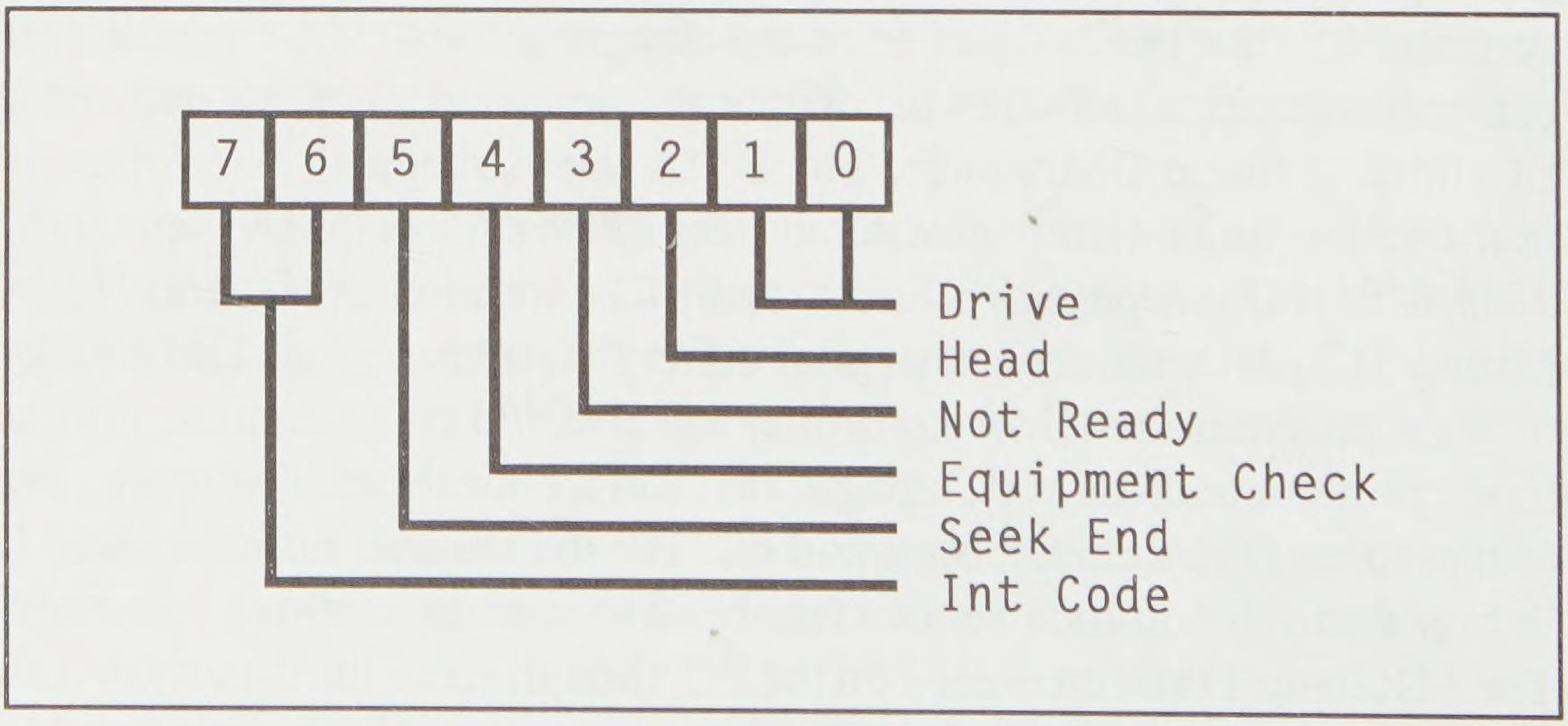


Figure 21.3 *Status Register `0`.*

The second initialization function is *fdcReset()*. *fdcReset()* resets the FDC and sends a *Specify* command to the FDC using the parameters you set up in your call to *fdcSetDriveParms()*. You should always call *fdcReset()* after you call *fdcSetDriveParms()* and before you send any other commands to the FDC. You also should call *fdcReset()* if you encounter any errors from the FDC.

FDC Commands

The FDC accepts a total of 15 commands, but I will only cover the most important ones. This section describes just the FDC commands; the commands may also return status bytes. The most important status bytes are Status Registers *0*, *1*, *2*, and *3* and the next section describes them in detail.

Each command consists of one or more bytes that *fdcSendCMD()* (Listing 21.3) sends to the FDC data register. Listing 21.3 also contains the functions that implement the various FDC commands described below. The code stores any status bytes in the global variable *Results*. Listing 21.4 contains the prototypes for the functions in Listing 21.3 and the constants that represent the various bits in registers and status bytes.

fdcSenseDrive() issues a Sense Drive Status command to request the current status of the specified drive. This command returns only one byte of information: Status Register *3*.

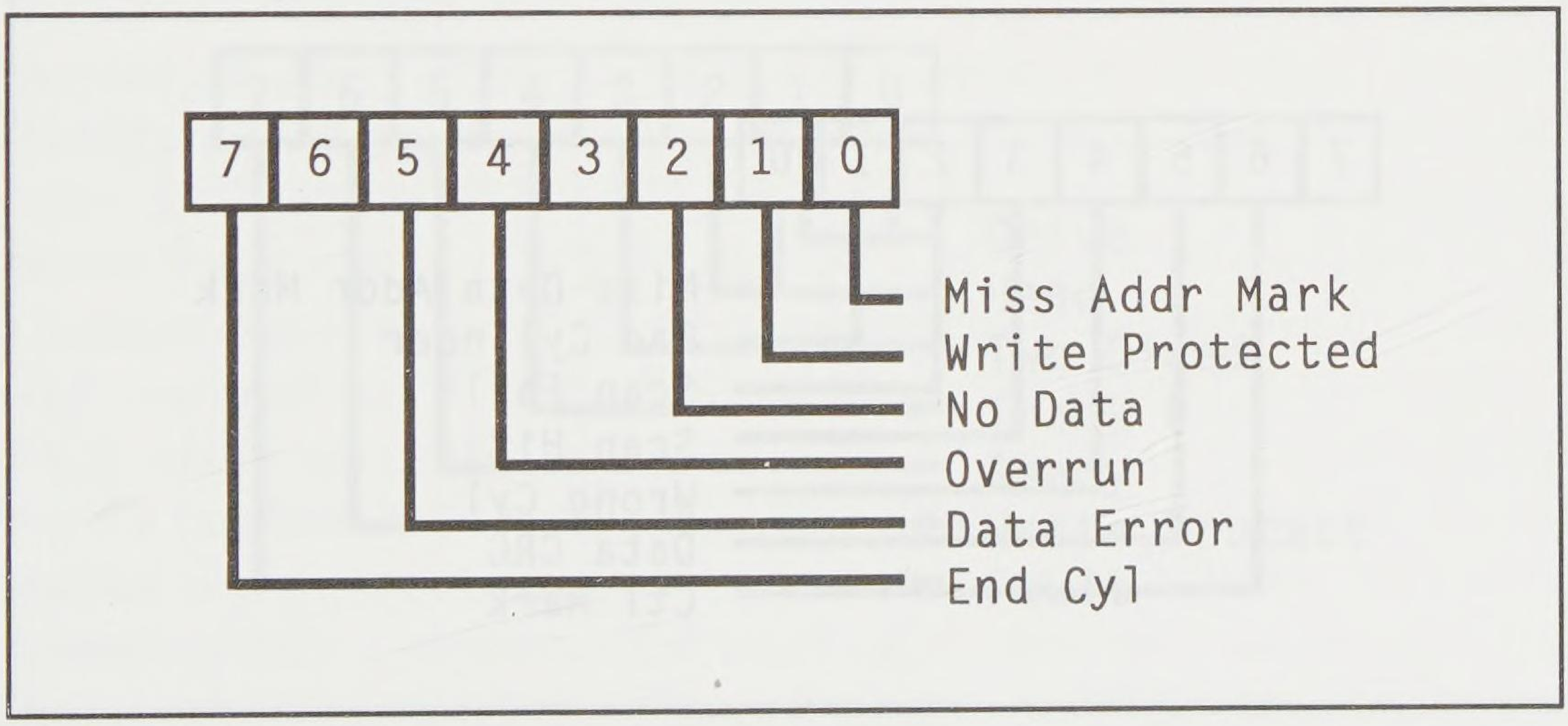


Figure 21.4 *Status Register* 1.

`fdcSenseInterrupt()` issues a Sense Interrupt Status command and should be called after a Seek or Recalibrate command generates its terminating interrupt. This command returns Status Register *0* and a byte containing the current position (cylinder number) of the head.

`fdcRecalibrate()` causes the FDC to move the drive head for the specified drive back to track zero. The function waits for an interrupt from the FDC (signifying that the command is complete) and then calls `fdcSenseInterrupt()` to get the status of the command.

`fdcSeek()` positions the drive head over a specific cylinder. As with `fdcRecalibrate()`, `fdcSeek()` uses `fdcSenseInterrupt()` to retrieve the status of the command after it completes.

The Read Data FDC command reads one or more sectors of information from a single cylinder. This command can read all the data from both sides of a diskette in a single operation. The function `fdcReadSectors()` shows how to use the Read Data command. This command can also read in a partial sector, because the command will end when the DMA controller has finished requesting data. This command returns seven bytes of data. The first three bytes are Status Registers *0*, *1*, and *2*. The last four bytes are the address mark (cylinder number, head number, sector number, and bytes-per-sector) of the last sector processed.

You can use `fdcReadSectors()` to figure out whether the drive contains a diskette. Simply read sector one on track zero, head zero. A successful read means that a disk is in the drive. If an error occurs, then

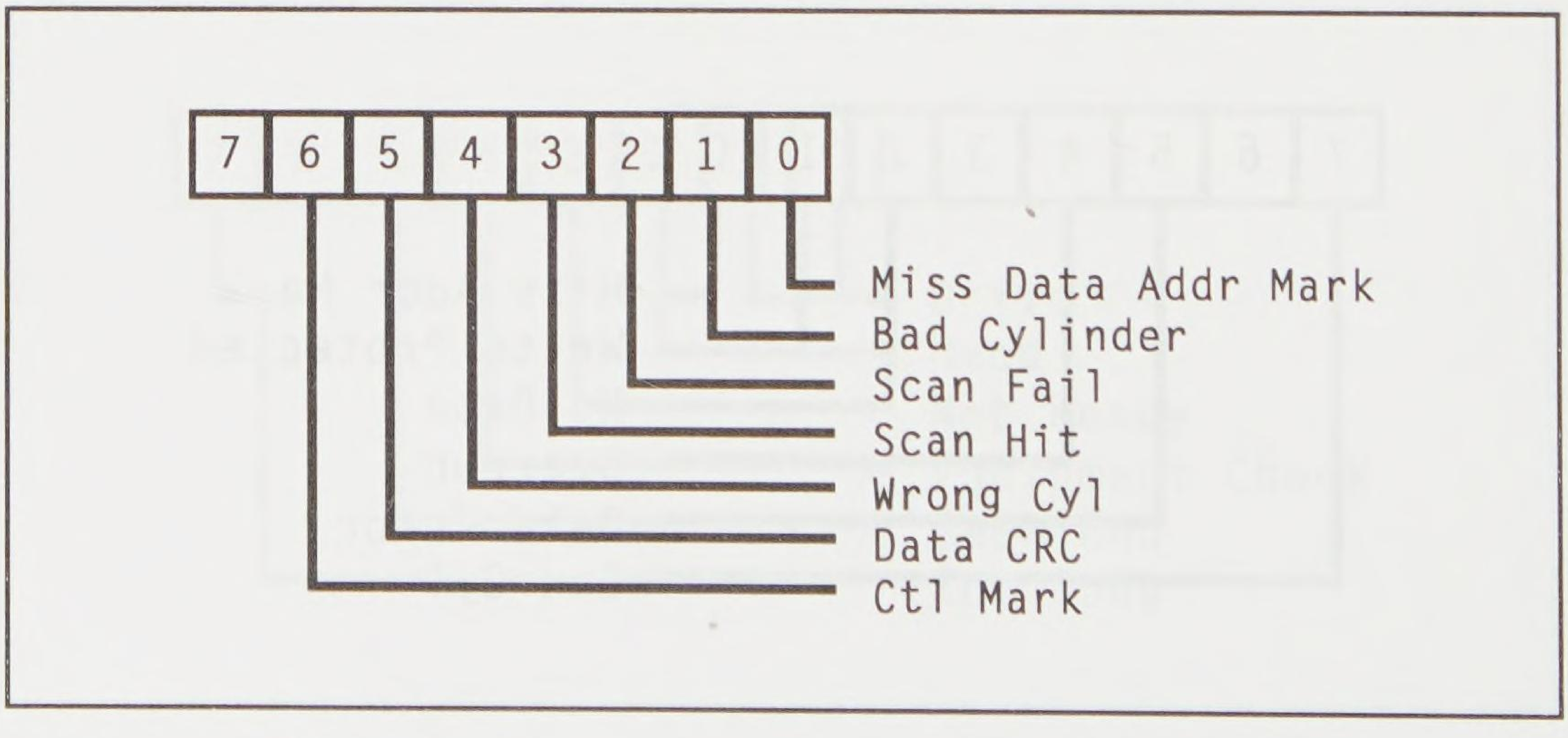


Figure 21.5 *Status Register 2.*

either the drive contains no disk, or the disk it contains is unreadable. This is a handy test to avoid an unnecessary "Abort, Retry, Ignore?" error.

The Write Data command is analogous to the Read Data command. It writes data instead of reading it; otherwise the steps are the same as with Read Data. `fdcWriteSectors()` performs these steps and returns the same seven bytes of information as `fdcReadSectors()`.

The Format Track command formats a single track. To format a complete cylinder, you must format each track on that cylinder with separate Format Track commands. The data passed to the FDC by the DMA controller for this command consists of four bytes (the address mark) for each sector to be created: the cylinder number, the head number, the sector number, and the bytes per sector.

Result Data

The following paragraphs describe the fields which make up the four status bytes that various FDC commands can return.

Status Register *0* (Figure 21.3) provides some general information about the result of an FDC command. Bits *0* and *1* specify the drive this status is for. The FDC stores the head number used in the command in bit *2*. Bit *3* (Not Ready) is turned on either to indicate that you attempted I/O before the drive was ready or you attempted I/O on the wrong side of a single-sided floppy. Bit *4* (Equipment Check) signals a fault signal

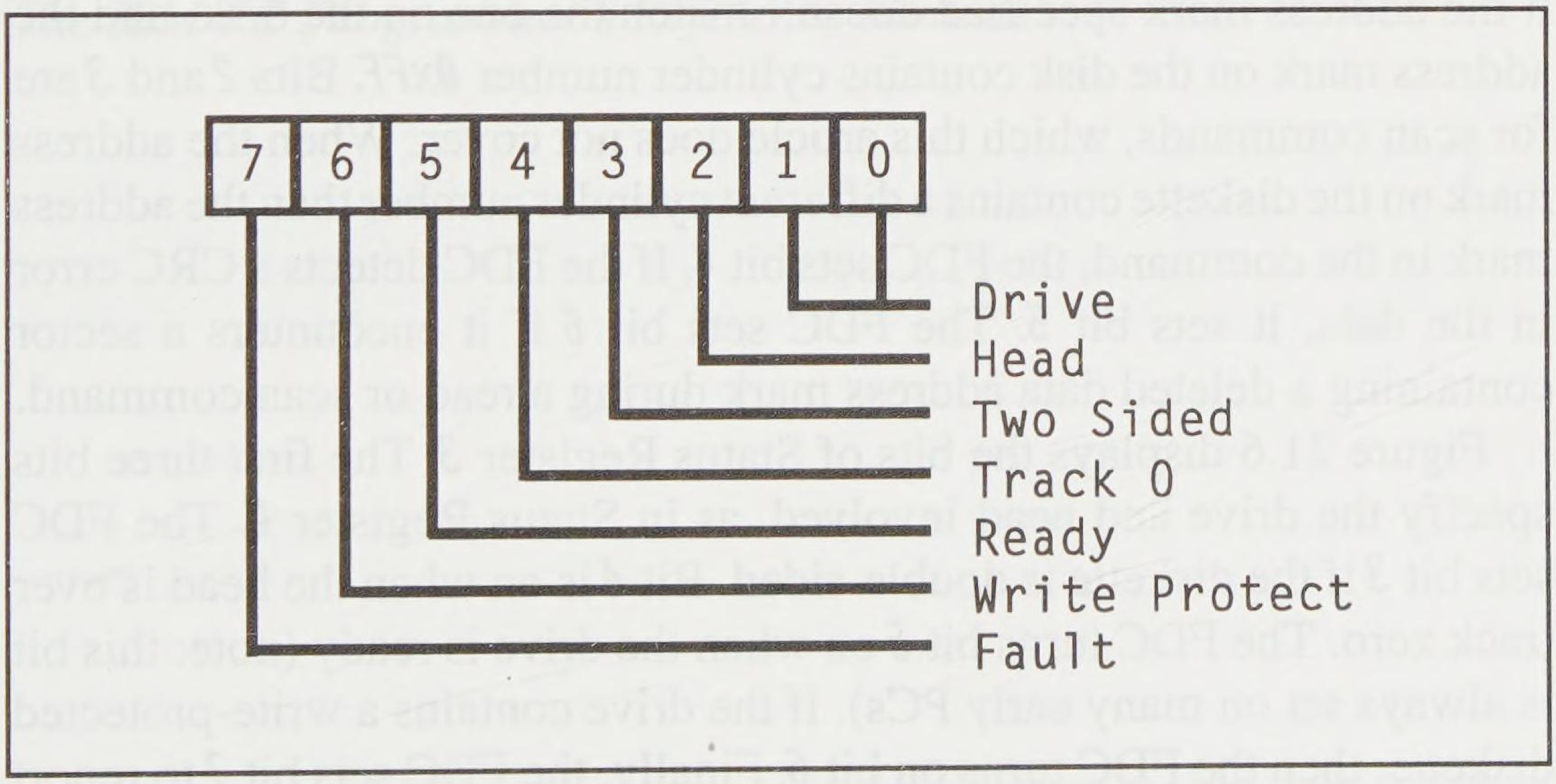


Figure 21.6 ***Status Register*** **3.**

from the FDC or else a recalibrate command could not locate track zero (either there was no disk in the drive or the drive has a serious problem). The FDC sets bit *5* (Seek End) to show that a seek or recalibrate command completed. Finally, bits *6* and *7* show the reason for this interrupt, as follows:

`00`	`Normal termination of the previous command`
`01`	`The previous command could not be completed`
`10`	`The last issued command was invalid`
`11`	`The command failed because the drive ready signal changed during command execution`

Figure 21.4 shows the layout of Status Register *1*. When the FDC can't find the desired address mark on the track, it sets bit *0*. If you try to write to or format a write-protected disk, the FDC sets bit *1*. If the FDC can't find the sector number (remember the sector number is just one of the four bytes in an address mark) of the desired address mark on the track, it sets bit *2*. Bit *3* is unused and is always *0*. If the DMA does not service the FDC fast enough during a read or write, the FDC sets bit *4*. Bit *5* is set if the FDC detects a CRC error in the address mark or the data of one of the processed sectors. A corrupted or damaged floppy disk can cause this error. Bit *6* is unused and is always *0*. Finally, attempting to access a sector beyond the final sector in a cylinder causes the FDC to set bit *7*.

Figure 21.5 shows the layout of Status Register *2*. Bit *0* is set if the FDC cannot locate an address mark on the cylinder. The FDC sets bit *1* if the address mark specified doesn't match the one on the disk, and the address mark on the disk contains cylinder number *0xFF*. Bits *2* and *3* are for scan commands, which this article does not cover. When the address mark on the diskette contains a different cylinder number than the address mark in the command, the FDC sets bit *4*. If the FDC detects a CRC error in the data, it sets bit *5*. The FDC sets bit *6* if it encounters a sector containing a deleted data address mark during a read or scan command.

Figure 21.6 displays the bits of Status Register *3*. The first three bits specify the drive and head involved, as in Status Register *0*. The FDC sets bit *3* if the diskette is double-sided. Bit *4* is on when the head is over track zero. The FDC turns bit *5* on when the drive is ready (note: this bit is always set on many early PCs). If the drive contains a write-protected diskette, then the FDC turns on bit *6*. Finally, the FDC sets bit *7* to report a fault.

Conclusion

Code that programs the FDC directly is difficult to debug because problems are often related to timing; stepping through code in a debugger often misses such problems. The easiest way to debug functions such as these is to build small test programs, such as the one in Listing 21.5, to test portions of these commands. The example in Listing 21.5 simply reads the first sector off the diskette in drive A, and writes it out to a file.

These routines give you the basic building blocks for asynchronous floppy disk I/O. The general idea is to modify the function `fdcWaitForInt()` to do something more useful than sitting in a loop. For example, if you are using a multi-threading package, you could make `fdcWaitForInt()` yield control to other threads each time it checks to see if the interrupt has occurred. Of course, you also have to make sure that no other threads try to start a floppy command before the current one has finished. Asynchronous I/O is more work than using the standard BIOS routines, but it can free your program from waiting around on slow floppy drives.

Bibliography

Thom Hogan, *The Programmer's PC Sourcebook*, Microsoft Press.

The AT&T PC 6300 Programmer's Reference Manual.

The IBM PC Programmer's Reference Manual.

Listing 21.1 delay.asm

```
;
; This file contains the routines to perform a 10ms delay using the system timer.
;
     .MODEL  SMALL, C

     .DATA
;
; Video segment and offset used in delay logic
;
video   LABEL   DWORD
vid_seg DW    0
vid_off DW    0

     .CODE

     PUBLIC  fdcDelay, calibrateDelay
;
; Perform a delay to allow the FDC Controller to catch up.
; This process works by waiting for the video memory,
; which is located on the system bus
;
fdcDelay PROC
     push   es
     les  bx, video
     mov  al, es:[bx]
     mov  es:[bx], al
     pop  es
     ret
fdcDelay ENDP
```

Listing 21.1 *continued*

```
;
; Determine the address to use for the delay operation.
; For color text modes, use the second video page in
; order to avoid creating snow on a CGA.
;
calibrateDelay PROC
    push  bp
    mov  ah, 0fh     ; get curent video mode
    int  10h
    cmp  al, 7       ; monochrome display?
    je   mono        ;  yes, set the address accordingly
    mov  vid_seg, 0B800h ; Color address == 0xB800:((page+1)%2)*4000
    xor  ax, ax      ; AX == offset
    mov  bl, bh      ; mov page number to BL
    inc  bl
    and  bl, 01h         ; use odd/even pages due to alternate text modes
    jz   page0
    mov  ax, 4000        ; use page 1
page0:
    mov  vid_off, ax
    jmp  SHORT done
mono:
    mov  vid_off, 0    ; Monochrome address == 0xB000:0000
    mov  vid_seg, 0B000h
done:
    pop  bp
    ret
calibrateDelay ENDP

    END

; End of File
```

Listing 21.2 `delay.h`

```
/*
**  Function prototypes for the Delay functions
**  in delay.asm.
*/
void   calibrateDelay(void);
void   fdcDelay(void);

/* End of File */
```

Listing 21.3 `floppy.c`

```
/*
**  This file contains the various functions used to
**  control the Floppy Disk Controller (FDC).
*/
#include    <stdio.h>
#include    <dos.h>
#include    "floppy.h"
#include    "delay.h"

/* Global data */
byte Results[7];

/* Command Strings */
static byte SpecifyCmd[] = { FdcCmdSpecify, 0, 0 };
static byte SenseDriveCmd[] = { FdcCmdSenseDrive, 0 };
static byte SenseIntCmd[] = { FdcCmdSenseInt };
static byte RecalCmd[] = { FdcCmdRecalibrate, 0 };
static byte SeekCmd[] = { FdcCmdSeek, 0, 0 };
static byte ReadCmd[] = { FdcCmdRead, 0, 0, 0, 0, 2, 9, 0x2a, 0xff };
static byte ReadTrackCmd[] = { FdcCmdReadTrack, 0, 0, 0, 0, 2, 9, 0x2a, 0xff };
static byte WriteCmd[] = { FdcCmdWrite, 0, 0, 0, 0, 2, 9, 0x2a, 0xff };
static byte FormatTrackCmd[] = { FdcCmdFormatTrack, 0, 2, 9, 0x50, 0xff };

/* Local data */
static byte StepRateTime = 0x0c,
            HeadUnloadTime = 0x0f,
            HeadLoadTime = 0x02;
static byte Drive;
static byte Cylinder = 0, Head = 0;
static unsigned BytesPerSector = 512;
static byte SectorsPerTrack = 9;
volatile static byte Interrupt = 0;
static void interrupt (*oldFdcInt)(void);

#define TRUE    1
#define FALSE   0

/* Process FDC Interrupt request */
void interrupt fdcInterrupt(void)
{
    Interrupt = TRUE;
    (*oldFdcInt)();
}
```

Listing 21.3 *continued*

```
/* Set local drive parameters value */
void fdcSetDriveParms(byte srt, byte hut, byte hlt)
{
    StepRateTime = srt;
    HeadUnloadTime = hut;
    HeadLoadTime = hlt;

    SpecifyCmd[1] = (StepRateTime << 4) | HeadUnloadTime;
    SpecifyCmd[2] = HeadLoadTime << 1;

    /* Setup interrupt hook, if not already set */
    if (oldFdcInt == NULL)
    {
        oldFdcInt = getvect(0x0e);
        setvect(0x0e, fdcInterrupt);
    }
}

/* Reset FDC interupt vector */
void fdcResetParms(void)
{
    if (oldFdcInt != NULL)
    {
        setvect(0x0e, oldFdcInt);
        oldFdcInt = NULL;
    }
}

/* Reset FDC and drives */
void fdcReset(void)
{
    /*
    ** Turn on the FDC's reset mode, disable DMA
    ** transfers, and turn off all drive motors.
    */
    fdcSetDOR(FdcDorReset | FdcDorDmaDisable);

    fdcDelay();

    /* turn off reset mode and enable DMA transfers */
    fdcSetDOR(FdcDorClearReset | FdcDorDmaEnable);

    /* Get the drive status  */
    fdcSenseDrive();

    /* Remind the controller what parameters we are using */
    fdcSpecify();
}
```

Listing 21.3 *continued*

```
/* Select, and turn on, the specified drive */
void fdcSelectDrive(byte drive)
{
    drive &= 0x03;    /* Limit drive numbers from 0 to 3 */
    Drive = drive;

    /* Set Command strings based on drive number */
    SenseDriveCmd[1] = (SenseDriveCmd[1] & ~0x03) | Drive;
    RecalCmd[1] = Drive;

    fdcSetDOR((FdcDorDriveOn << drive) | FdcDorDmaEnable | FdcDorClearReset | drive);
}

/* Specify drive parameters */
int fdcSpecify(void)
{
    fdcSendCmd(SpecifyCmd, sizeof(SpecifyCmd));
    return fdcResultPhase(FALSE, 0);
}

/* Sense Drive Status */
unsigned fdcSenseDrive(void)
{
    fdcSendCmd(SenseDriveCmd, sizeof(SenseDriveCmd));
    return fdcResultPhase(TRUE, 1);
}

/* Get information concerning the last drive interrupt */
unsigned fdcSenseInterrupt(void)
{
    fdcSendCmd(SenseIntCmd, sizeof(SenseIntCmd));
    return fdcResultPhase(TRUE, 2);
}

/* Recalibrate the drive */
unsigned fdcRecalibrate(void)
{
    Cylinder = Head = 0;

    fdcSendCmd(RecalCmd, sizeof(RecalCmd));

    fdcWaitForInt();
    return fdcSenseInterrupt();
}
```

Listing 21.3 *continued*

```
/* Seek the drive heads to the specified cylinder */
unsigned fdcSeek(int head, int cylinder)
{
    Cylinder = cylinder;
    Head = head;

    SeekCmd[1] = (head << 2) | Drive;
    SeekCmd[2] = cylinder;

    fdcSendCmd(SeekCmd, sizeof(SeekCmd));

    fdcWaitForInt();
    return fdcSenseInterrupt();
}

/*
** Read a number of sectors from the current track into
** the read buffer passed by the caller.
*/
unsigned fdcReadSectors(int sector, unsigned nsect, void* buffer)
{
    unsigned byteCount = nsect * BytesPerSector;

    /* Test for buffer fitting within a single DMA page */
    if (!dmaPageTest(buffer, byteCount))
        return -5;

    /* Setup DMA Controller */
    dmaSetup(DmaCmdRead, byteCount, buffer);

    ReadCmd[1] = (Head << 2) | Drive;
    ReadCmd[2] = Cylinder;
    ReadCmd[3] = Head;
    ReadCmd[4] = sector;

    fdcSendCmd(ReadCmd, sizeof(ReadCmd));
    return fdcResultPhase(FALSE, 7);
}
```

Listing 21.3 *continued*

```
unsigned fdcReadTrack(void* buffer)
{
    /* Test for buffer fitting within DMA boundries */
    if (!dmaPageTest(buffer, SectorsPerTrack * BytesPerSector))
          return -5;

    /* Setup DMA Controller */
    if (!dmaSetup(DmaCmdRead, SectorsPerTrack * BytesPerSector, buffer))
          return -5;

    ReadTrackCmd[1] = (Head << 2) | Drive;
    ReadTrackCmd[2] = Cylinder;
    ReadTrackCmd[3] = Head;

    fdcSendCmd(ReadTrackCmd, sizeof(ReadTrackCmd));
    return fdcResultPhase(FALSE, 7);
}

/*
** Write a number of sectors from the current track from
** the read buffer passed by the caller.
*/
unsigned fdcWriteSectors(int sector, unsigned nsect, void* buffer)
{
    unsigned byteCount = nsect * BytesPerSector;

    /* Test for buffer fitting within a single DMA page */
    if (!dmaPageTest(buffer, byteCount))
        return -5;

    /* Setup DMA Controller */
    dmaSetup(DmaCmdWrite, byteCount, buffer);

    WriteCmd[1] = (Head << 2) | Drive;
    WriteCmd[2] = Cylinder;
    WriteCmd[3] = Head;
    WriteCmd[4] = sector;

    fdcSendCmd(WriteCmd, sizeof(WriteCmd));
    return fdcResultPhase(FALSE, 7);
}
```

Listing 21.3 *continued*

```
/* Format the current track */
unsigned fdcFormatTrack(void* buffer)
{
   /* Test for buffer fitting within DMA boundries */
   if (!dmaPageTest(buffer, SectorsPerTrack * 4))
      return -5;

   /* Setup DMA Controller */
   if (!dmaSetup(DmaCmdWrite, SectorsPerTrack * 4, buffer))
      return -5;

   FormatTrackCmd[1] = (Head << 2) | Drive;
   FormatTrackCmd[2] = Head;

   fdcSendCmd(FormatTrackCmd, sizeof(FormatTrackCmd));
   return fdcResultPhase(FALSE, 7);
}

int fdcSendCmd(byte* cmd, unsigned cmdlen)
{
   int rqmTimeout, i;

   /* Send the command to the FDC */
   for (i = 0 ; i < cmdlen ; ++i)
   {
      fdcDelay();

      /* Wait for the RQM flag */
      rqmTimeout = 0;
      while ((fdcGetStatus() & FdcMsrRequest) != FdcMsrRequest)
      {
         if (++rqmTimeout > 1000)
            return -1;
      }

      /* Check that the FDC is ready for input */
      if ((fdcGetStatus() & FdcMsrDirection) != 0)
         return -2;

      fdcWriteCmd(*cmd);
      ++cmd;
   }

   return 0;
}
```

Listing 21.3 *continued*

```
/* Wait for an interrupt */
void fdcWaitForInt(void)
{
    while (!Interrupt)
        ;

    Interrupt = FALSE;
}

/* Wait for result data */
int fdcResultPhase(int immediate, int nres)
{
    /* wait for interrupt if not immediate result phase */
    if (!immediate)
        fdcWaitForInt();

    /* Turn off the interrupt flag */
    Interrupt = FALSE;

    /* Input results from the FDC */
    if (immediate || (fdcGetStatus() & FdcMsrFdcBusy) != 0)
    {
        /* operation is completed, get FDC result bytes */
        int i, rqmTimeout;

        for (i = 0 ; i < nres ; ++i)
        {
            fdcDelay();

            /* Wait for the RQM flag */
            rqmTimeout = 0;
            while ((fdcGetStatus() & FdcMsrRequest) != FdcMsrRequest)
            {
                if (++rqmTimeout > 1000)
                    return -1;
            }

            /* Check that the FDC is ready for input */
            if ((fdcGetStatus() & FdcMsrDirection) == 0)
                return -2;

            Results[i] = fdcGetResult();
        }
```

Listing 21.3 *continued*

```
        /* Make sure that FDC has completed sending data */
        if ((fdcGetStatus() & FdcMsrFdcBusy) != 0)
            return -3;
    }
    else
        /* Catch all hiddin interrupts */
        while (fdcSenseInterrupt() != 0);

    return 0;
}

/*
** Test to see if buffer fits within a single DMA page, return TRUE is ok
*/
int dmaPageTest(byte far* buf, unsigned length)
{
    unsigned page1, page2;

    page1 = (unsigned)(ptrToLong(buf) >> 16) & 0x000f;
    page2 = (unsigned)(ptrToLong(buf + length) >> 16) & 0x000f;

    return (page1 == page2);
}

/* Setup the DMA controller */
int dmaSetup(int fcn, unsigned nbytes, void far* addr)
{
    unsigned long ptr = ptrToLong(addr);

    disable();

    /* Set DMA mode and reset the first/last flip-flop */
    outportb(DmaModePort, fcn);
    fdcDelay();
    outportb(DmaFFPort, 0);
    fdcDelay();

    /* Set DMA Address */
    outportb(DmaAddrPort,  ptr & 0x000000ffL);
    fdcDelay();
    outportb(DmaAddrPort, (ptr & 0x0000ff00L) >> 8);
    fdcDelay();
    outportb(DmaPagePort, (ptr & 0x000f0000L) >> 12);
    fdcDelay();
```

Listing 21.3 *continued*

```
    /* Set transfer word count */
    --nbytes;
    outportb(DmaWcPort, nbytes & 0x00ff);
    fdcDelay();
    outportb(DmaWcPort, (nbytes & 0xff00) >> 8);
    fdcDelay();

    enable();

    /* Start the DMA channel */
    outportb(DmaMaskPort, DmaFdcChannel);

    return 0;
}

/* End of File */
```

Listing 21.4 `floppy.h`

```
/*
** Definitions for the Floppy Disk Controller (FDC) ports and commands.
*/

typedef unsigned char byte;

/* FDC Ports addresses */
#define FdcOutputRegister   (0x03f2)
#define FdcStatusRegister   (0x03f4)
#define FdcCommandRegister  (0x03f4)
#define FdcDataRegister     (0x03f5)

/* FDC data macros */
#define fdcSetDOR(value)    outportb(FdcOutputRegister, (value))
#define fdcWriteCmd(value)  outportb(FdcDataRegister, (value))
#define fdcGetStatus()      inportb(FdcStatusRegister)
#define fdcGetResult()      inportb(FdcDataRegister)

/* FDC Digital Output Register (DOR) Commands */
#define FdcDorDrive0Select  (0x00)
#define FdcDorDrive1Select  (0x01)
#define FdcDorDrive2Select  (0x02)
#define FdcDorDrive3Select  (0x03)
#define FdcDorReset         (0x00)
#define FdcDorClearReset    (0x04)
#define FdcDorDmaDisable    (0x00)
#define FdcDorDmaEnable     (0x08)
#define FdcDorDriveOn       (0x10)

/* FDC Main Status Register (MSR) bit flags */
#define FdcMsrDrive0Seek    (0x01)   /* Drive 0 is seeking */
#define FdcMsrDrive1Seek    (0x02)   /* Drive 1 is seeking */
#define FdcMsrDrive2Seek    (0x04)   /* Drive 2 is seeking */
#define FdcMsrDrive3Seek    (0x08)   /* Drive 3 is seeking */
#define FdcMsrFdcBusy       (0x10)   /* Read/Write in progress */
#define FdcMsrExecMode      (0x20)   /* non-DMA Execute phase */
#define FdcMsrDirection     (0x40)   /* set if FDC->CPU */
#define FdcMsrRequest       (0x80)   /* set if ready for master to read */

/* FDC Commands */
#define FdcCmdSpecify       (0x03)   /* Set FDC parameters */
#define FdcCmdSenseDrive    (0x04)   /* Get drive state */
#define FdcCmdSenseInt      (0x08)
#define FdcCmdRecalibrate   (0x07)
#define FdcCmdSeek          (0x0f)
#define FdcCmdRead      (0x06 | 0x80)   /* Read, Multitrack */
#define FdcCmdReadTrack     (0x02)
#define FdcCmdWrite     (0x05 | 0x80)   /* Write, Multitrack */
#define FdcCmdFormatTrack   (0x0d)
```

Listing 21.4 *continued*

```
/* DMA Defines */
#define DmaFdcChannel   (0x02)
#define DmaMaskPort     (0x0a)
#define DmaModePort     (0x0b)
#define DmaFFPort       (0x0c)
#define DmaAddrPort     (2 * DmaFdcChannel)
#define DmaWcPort       (DmaAddrPort + 1)
#define DmaPagePort     (0x81)
#define DmaCmdRead      (0x44 | DmaFdcChannel)
#define DmaCmdWrite     (0x48 | DmaFdcChannel)

/* Convert a far pointer to a long */
#define ptrToLong(ptr)  ((((unsigned long)FP_SEG(ptr)) \
<< 4) + FP_OFF(ptr))

/* Global data */
extern byte Results[7];

/* Function protoypes */
void fdcSetDriveParms(byte srt, byte hut, byte hlt);
void fdcResetParms(void);
void fdcReset(void);
void fdcSelectDrive(byte drive);
int fdcSpecify(void);
unsigned fdcSenseDrive(void);
unsigned fdcSenseInterrupt(void);
unsigned fdcRecalibrate(void);
unsigned fdcSeek(int head, int cylinder);
unsigned fdcReadSectors(int sector, unsigned nsect, void* buffer);
unsigned fdcReadTrack(void* buffer);
unsigned fdcWriteSectors(int sector, unsigned nsect, void* buffer);
unsigned fdcFormatTrack(void* buffer);
int fdcSendCmd(byte* cmd, unsigned cmdlen);
void fdcWaitForInt(void);
int fdcResultPhase(int immediate, int nres);
int dmaPageTest(byte far* buf, unsigned length);
int dmaSetup(int fcn, unsigned nbytes, void far* addr);

/* End of File */
```

Listing 21.5 *Test code for the FDC routine in* `floppy.c`.

```
#include        <stdio.h>
#include        "floppy.h"
#include        "delay.h"

static char buffer[512];

int
main(void)
{
        unsigned rc;

        /* Setup FDC */
        calibrateDelay();
        fdcSetDriveParms(0x0c, 0x0f, 0x02);
        fdcReset();

        /* Select drive 0, and recalibrate it */
        fdcSelectDrive(0);
        rc = fdcRecalibrate();
        printf("fdcRecalibrate() == %u\n", rc);

        /* Read in sector 1 from head 0 */
        rc = fdcSeek(0, 0);
        printf("fdcSeek(0, 0) == %u\n", rc);
        rc = fdcReadSectors(1, 1, buffer);
        printf("fdcReadSectors(1, 1, buffer) == %u\n", rc);

        fdcResetParms();

        {
                FILE *fd;

                fd = fopen("tmp.dat", "wb");
                fwrite(buffer, 1, sizeof(buffer), fd);
                fclose(fd);
        }

        return 0;
}

/* End of File */
```

Chapter 22

Writing for the PC Speaker

Robert D. Bybee

The IBM PC speaker interface was originally designed to play only simple tones. Using an interesting software technique, applications can now produce reasonably high quality voice and music through the PC speaker. Some commercially available programs (mostly games) already take advantage of this method. Now your programs can do the same.

Physics Review

A speaker produces sound by moving air back and forth many times per second. The PC speaker interface consists of some digital hardware, driving an amplifier which moves the speaker's cardboard cone. To produce a tone of, say, 262Hz (middle C on a piano), the interface must move the speaker back and forth 262 times per second.

The PC Implementation

The PC uses two different methods to move the speaker. One of the timers on the motherboard (timer two in the 8253 chip, or its equivalent) can be programmed to generate a square wave of various frequencies. Setting bit *0* of output port *61H* to *1* sends the square wave to the speaker. Most programs, including the BIOS, use this method to generate a "beep" tone.

Bit *1* of the same port (*61H*) drives the speaker directly. When this bit is set to *1*, the circuit applies a constant voltage to the speaker cone, forcing it in one direction. When this bit is *0*, the speaker has no voltage applied and returns to its normal, resting position. By turning this bit on and off rapidly, software can generate a tone of any desired frequency.

To compare the two methods, imagine a program that generates a 1,000Hz tone for one second. Using the programmable timer, software would set the speaker to generate 1,000Hz, enable it by setting port *61H* bit *0*, do nothing for one second, then clear port *61H* bit *0*. Alternatively, using

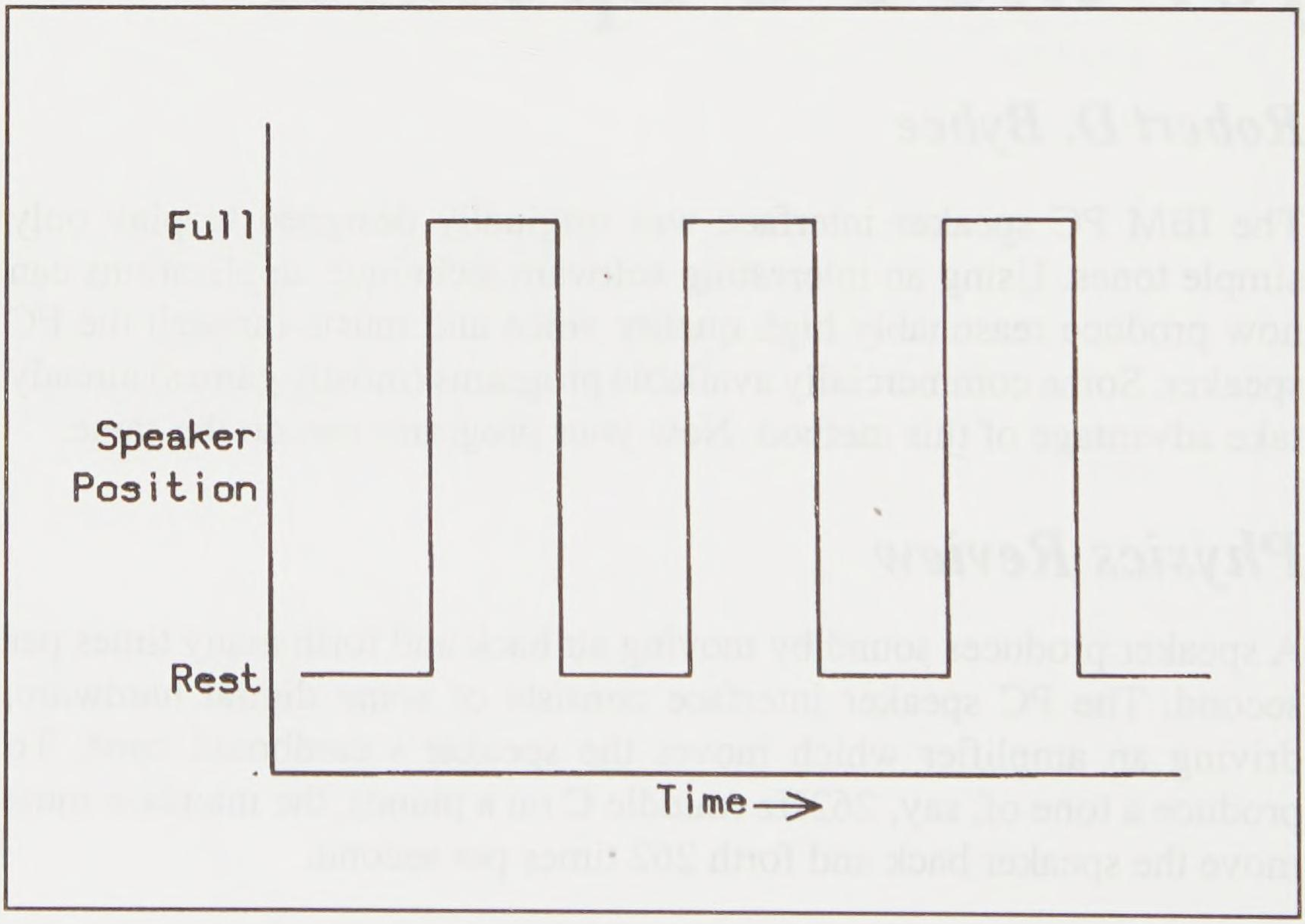


Figure 22.1 *Digital waveform.*

port *61H* bit *1*, software would have to set the bit, wait .0005 seconds, clear it, wait .0005 seconds, and repeat this loop 1000 times. This process is a lot more difficult, especially since a short delay time like .0005 seconds must be generated using a carefully timed set of instructions. The number of instructions varies depending on CPU type, clock speed, memory speed, and so on.

Digital vs. Analog Waveforms

Both of these techniques can generate a square wave, such as the one in Figure 22.1. When reproducing a square wave, the speaker position is either "on" (full extension) or "off" (resting). There is no intermediate position. But to play recognizable voice or high quality music, the hardware must be capable of doing much more.

Figure 22.2 shows a portion of a voice waveform. It contains many points between the two extremes of the square wave. Some computers use a Digital-to-Analog Converter (DAC) to drive the speaker. These

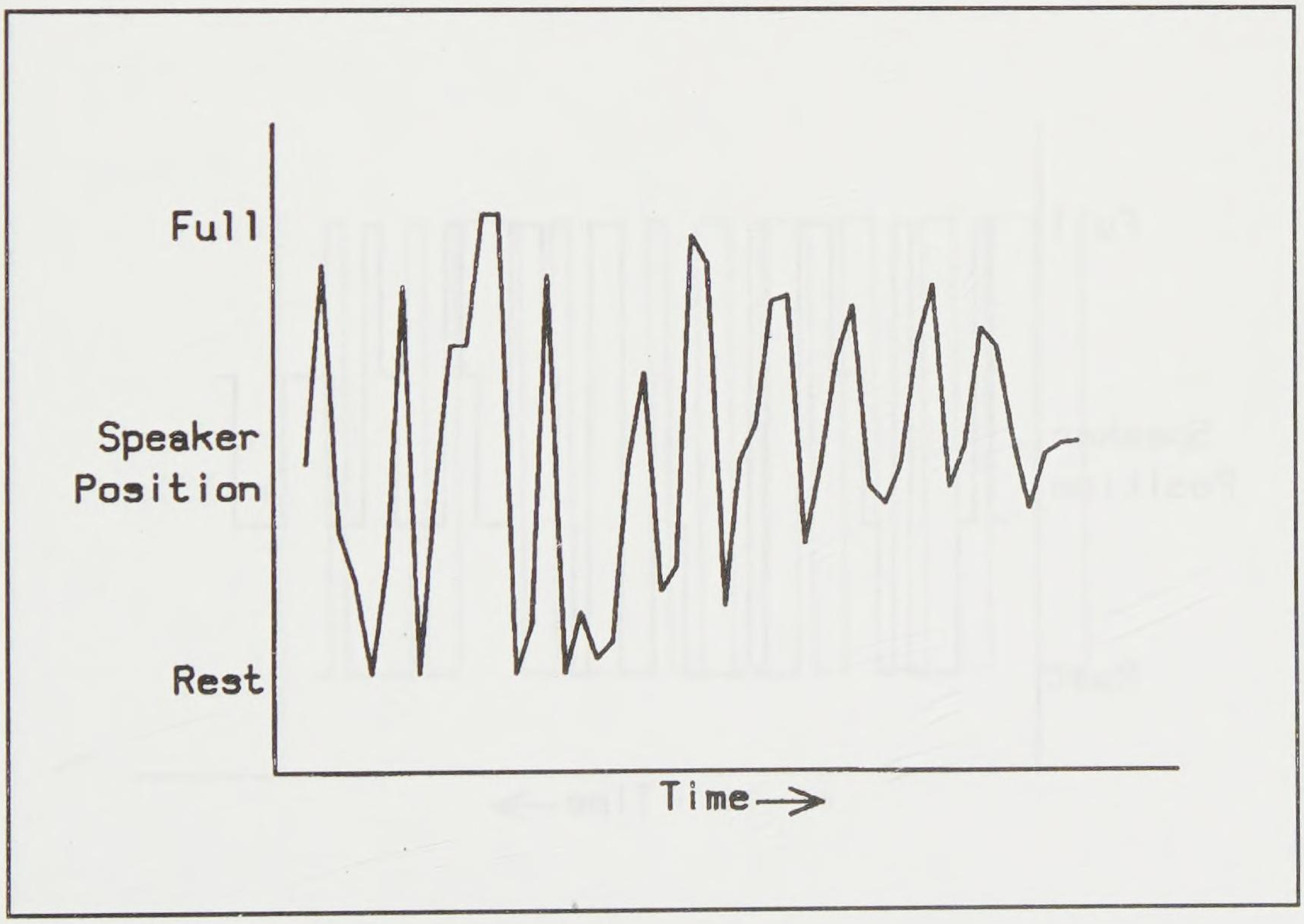


Figure 22.2 ***Analog waveform.***

systems can more closely approximate the intermediate values that a voice waveform requires. A Macintosh, for example, uses an 8-bit DAC, capable of producing 256 different speaker positions. A compact-disc player uses 16 bits for 65,536 possible positions and very high quality.

Figure 22.3 illustrates the same voice data used in Figure 22.2, but "quantizes" it into four discrete levels (as a 2-bit DAC would). As a digital system uses fewer bits to represent the waveform, its output has fewer distinct levels and becomes less like the original analog voltage. The PC has the equivalent of a 1-bit DAC, with only two positions: on and off. Running the voice waveform through a 1-bit DAC looks like Figure 22.4 and sounds like pure distortion.

A Mechanical Integrator

The CPU is many times faster than the speaker. When the CPU sets port *61H* bit *1*, the speaker tends to be driven outward. But because the speaker is a mechanical device and has inertia, it does not attain its maximum excursion immediately.

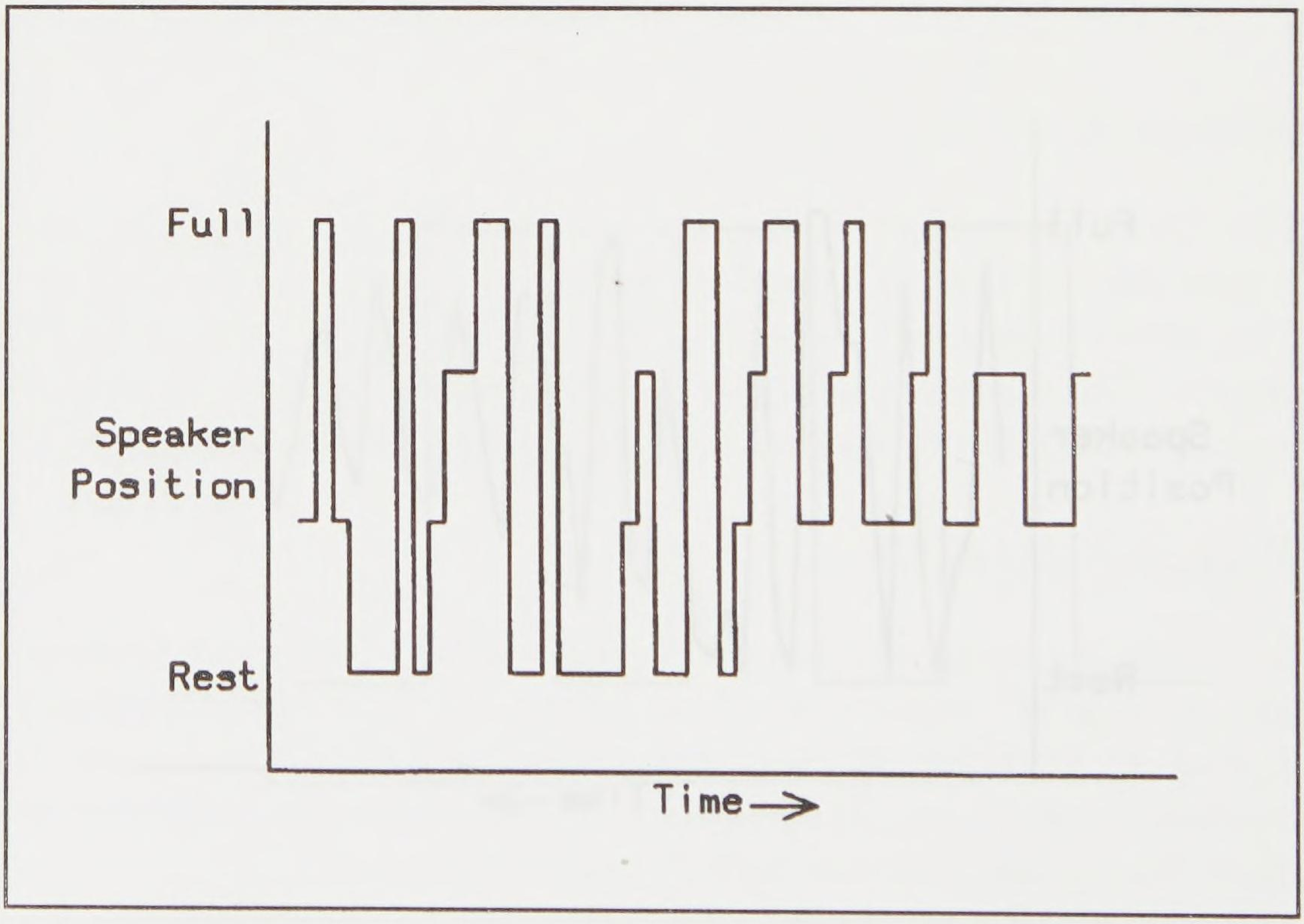


Figure 22.3 *A voice waveform run through a 2-bit DAC and "quantized" into four discrete levels.*

Suppose that before the speaker reaches its maximum position, the CPU clears port *61H* bit *1*. The driving voltage disappears, and the speaker tries to return to its original position. Now, if the CPU sets *and* clears this bit very rapidly, the speaker experiences approximately half the driving voltage it did before (since the bit is on only half the time). The resulting position falls halfway between the two extreme points.

This method, called pulse-width modulation, can be used to set the speaker anywhere between its "off" and "on" positions. By varying the ratio of ones and zeros fed to port *61H* bit *1*, a program can set the speaker position with fairly good accuracy. The speaker acts as a mechanical integrator and follows the average voltage fed to it, since it is too slow to respond to every bit the CPU writes.

How Many Bits?

In any digitizing application, the more bits the program uses, the better. To maximize this algorithm's quality, the code should try to force as many bits per second to the speaker as it possibly can.

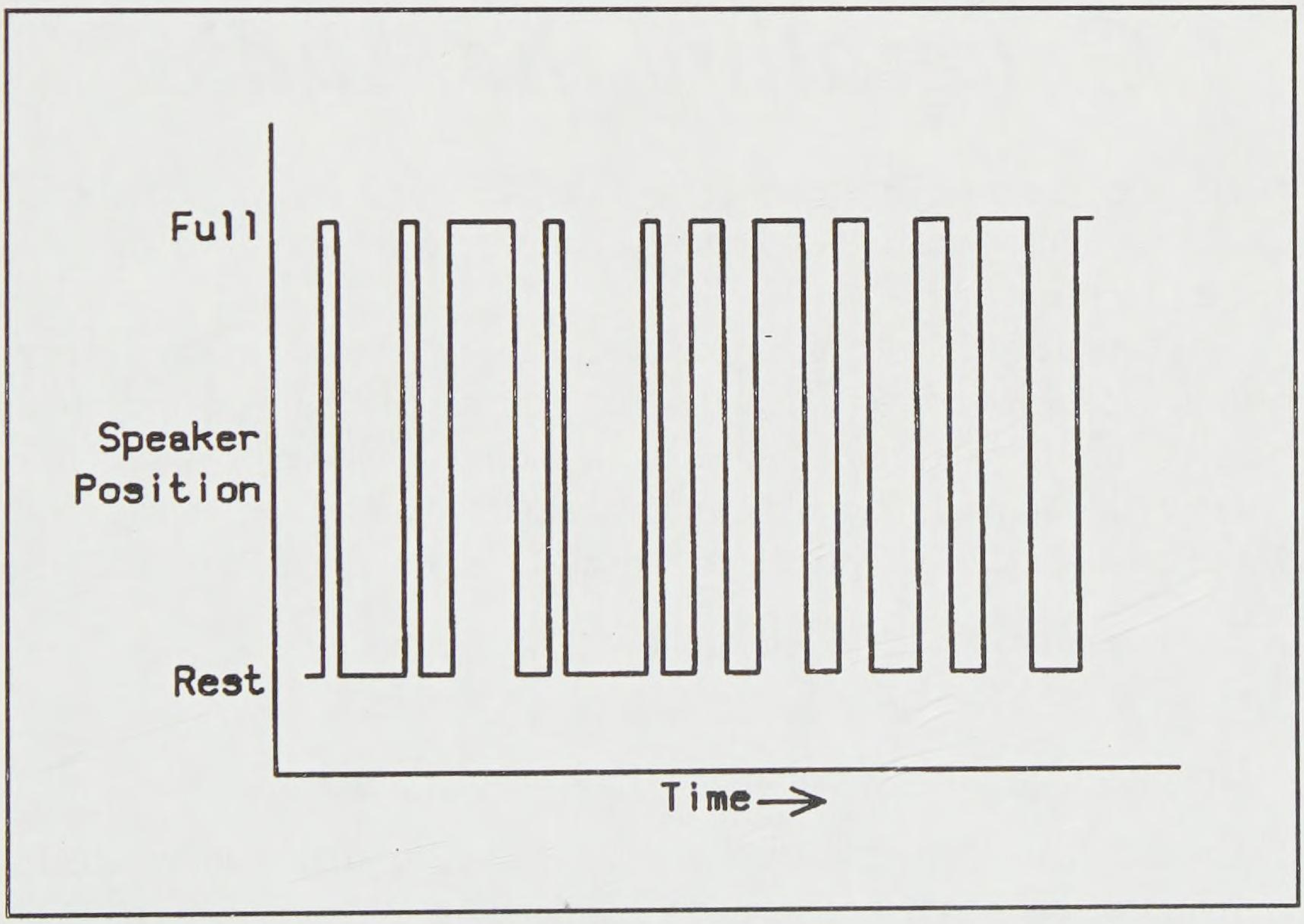


Figure 22.4 *A voice waveform run through a 1-bit DAC and "quantized" into two distinct levels.*

Suppose I recorded a file of digitized audio, using a sampling rate of 8000 samples per second. If I recorded this file with an 8-bit Analog-to-Digital Converter (ADC), each sample could have 256 possible levels. To reproduce the sound, the CPU must fetch a new sample from the file every 1/8000 second (125 microseconds), and feed some ones and zeros to the speaker so that the speaker moves to its new position.

If the sample's value were `00`, all bits sent to the speaker would be zeros. If the sample were `FFh`, all bits would be ones. For a sample of `80h`, the bits would alternate between ones and zeros, positioning the speaker somewhere in the middle of its range.

Ideally, the program should push 255 bits to the speaker for each sample. The number of ones sent in each sample would equal the value of the sample. Sample `00h` would have no ones, `80h` would have 128 ones (evenly distributed among the zeros, please). In sample `FFh`, all 255 bits would be ones. Unfortunately, I haven't found a method of pushing 255 bits out to the speaker, 8,000 times per second. To do so, I would have

Generating the Audio

The `TOSPKR` program illustrates how to play an audio file through the PC speaker with moderately good quality. But where did that file come from in the first place?

To generate a file with music or sound effects, you need a development system capable of converting audio into a digital file. Several companies build PC-bus boards for telephone applications such as voice-mail. These can record (and play) telephone-quality digitized audio, which has about 3,000Hz bandwidth, and is perfectly adequate for PC-speaker applications.

Some sources for these telephone-interface boards are:

- Innovative Technology Inc., Roswell, GA.
- Dialogic Corporation, Parsippany, NJ.

There are others. Most electronics and software journals contain ads for companies that sell voice response applications systems. All of these systems use some kind of digital audio board. You can also use any to write a new value to port *61H*, (8000 x 255) times per second, or one every 0.49 microseconds.

Even if new CPUs were that fast, the program should run correctly on all PCs, not just the fastest ones. The program should adapt itself to the speed of the PC it is running on, and push as many bits to the speaker as that PC is capable of doing.

Assembly Required

The fastest way to send bits to the speaker is in a series of instructions:

```
mov   al,ch
out   dx,al
mov   al,cl
out   dx,al
.
.
```

standard analog-to-digital converter board, if it can capture audio levels at about 8,000 samples per second.

ITI's board records 8,000, 8-bit samples per second, but encodes its files using the telephone industry's Mu-Law compression algorithm. Listing 22.2, *U2BIN.C*, converts these files into the offset binary files used by *TOSPKR*. *U2BIN* uncompresses the Mu-Law data and clips it to make it louder on the PC speaker. Any distortion introduced by this process is buried in the inherent PC speaker distortion.

Some of Dialogic's new boards can record Mu-Law data files. Their older boards use only Adaptive Delta Pulse Code Modulation (ADPCM), which reduces the storage requirements by recording only 6,000, 4-bit samples per second. ADPCM files can be converted into offset binary, but the method is more complex and the sound quality isn't as good. Other audio boards may record in a format different from either of these. Be sure you obtain information on the data format used by the audio board you plan to buy.

If `ch` is preloaded with a value that has bit *1* set, `cl` has the same value but with bit *1* clear, and `dx` contains *61H* (the port number), this code will slam bits to the speaker as fast as the CPU can run. The program can read from port *61H* to determine what other bits are currently set. The other bits in port *61H* are used to control other devices on the PC's system board and should be left alone.

The system's speed determines how many `mov/out` instruction pairs can be executed in 125 microseconds. The program finds this number experimentally, then assembles the proper number of opcodes in dynamically allocated memory. The program actually assembles 256 routines in this manner, so that for every byte value in the file, exactly one routine is called. Each of these routines takes 125 microseconds to execute, and drives the speaker port with a set of ones and zeros that moves the speaker to the proper position.

The TOSPKR *Program*

TOSPKR, shown in Listing 22.1, implements the algorithm. The program takes two command line arguments. The first parameter, the name of the audio data file to be played, is required. The second, optional argument is the number of bits to be fed to the speaker for each audio sample. If you already know your system's speed, you can provide this number on the command line. If not, the program will calculate it and you can enter it next time. *TOSPKR* takes about five seconds to calculate this value. (Other versions of the program could store this value in a file so that it can be read back automatically.)

The program's overall logic is:

1. Open the input file.
2. If the number of bits per sample is provided on the command line, use it; otherwise calculate it.
3. Create the 256 functions needed. Load pointers to these functions into an array.
4. If the file contains a list of audio segment lengths, read in that list. This information is used to pace the disk I/O.
5. Read the input file. For each byte of the file, call one of the 256 functions.

Bits Per Sample

The `best_rate()` routine determines how fast your system is, and therefore how many bits (per audio sample) it can send to the speaker. `best_rate()` begins with an estimate of 10 bits per sample and calls `test_time()` with that estimate. `test_time()` generates instructions that send 10 bits to the speaker, then executes those instructions `SAMPRATE` times. If the initial estimate is correct, this process will take one second (18.2 BIOS ticks) to complete.

Since the initial estimate is usually low, `best_rate()` doubles the estimate until `test_time()` takes at least one second to run. At that point the variable `bits1` indicates number of bits per sample, and `ticks1` tells the time it took (in BIOS ticks).

Now the program tries a higher estimate, using `bits2` which is five times the current `bits1` estimate. This calculation returns a larger number of ticks, called `ticks2`. Finally, the program finds the optimum number of bits per sample, using linear interpolation and the two-point equation of a line:

$$(y - y_1) = (x - x_1) * (y_2 - y_1) / (x_2 - x_1)$$

The $x$es are the number of bits, and ys are the elapsed time in ticks. The target y is 18.2 ticks (one second), and x is the number of bits necessary to create this y. Letting $y = 18.2$ and solving for x,

$$x = (18.2 - y_1) * (x_2 - x_1) / (y_2 - y_1) + x_1$$

The code plugs in `bits1` for x_1, `ticks1` for y_1, and so on. These calculations are done in floating-point, which doesn't cost much time since they are done only once. (See the sidebar, "A Borland Compiler Floating-Point 'Feature.'")

Interestingly enough, the optimum number of bits does not always increase as the system speed increases. A fast 80286 can push about 113 bits per sample; some 80386 machines can do only 65. This fact tells me that not all of the instructions gained speed as Intel built faster processors. The particular set of instructions in this code actually runs slower on a 386.

Generating Instruction Streams

Now that the program knows how many bits per sample it should play, it calls `generate_codes()` to create 256 executable functions in dynamically allocated memory. This may be the ultimate in self-modifying code, since it generates far more code dynamically than the program originally contained!

Each of the generated functions begins with the opcodes in the `ops_begin[]` array and ends with the opcodes in `ops_end[]`. In between, there are `bits_per_sample` copies of either the `ops_high[]` or `ops_low[]`

A Borland Compiler Floating-Point "Feature"

While testing the `TOSPKR` program, *Windows™/DOS Developer's Journal* editor Ron Burk discovered the program wouldn't run on one of his 80286-based systems. He narrowed the problem down to the floating-point expression that calculates `fbits`. The code simply wasn't storing a value into the result.

The problem turned out not to be in the program or in Ron's hardware. Because of the way Borland's C compilers do floating-point, this bug will show up in a small number of systems. If you distribute programs compiled with Turbo C or Borland C++, you should be aware of this problem and a work around for it.

These compilers allow you to select one of three floating-point options:

- `none` — Use this if your program contains no floating-point at all.
- `80x87` — Use this if you're positive that your program will always run on a system with a numeric coprocessor.
- `emulation` — When the program starts, it determines if there's a math coprocessor in the system. If so, it uses the chip. If not, it uses the floating-point emulation library.

The problem is caused by the method Borland uses to detect an 80x87 chip. During startup, the program makes a call to BIOS interrupt `0x11`, array. `ops_high[]` contains opcodes which push a *1* out to the speaker, and `ops_low[]` writes `0` to the speaker bit.

The distribution of ones and zeros is controlled by a "division by overflow" method. It uses a variable called `sum`, which begins at zero and is repeatedly augmented by the current `value` (0 to 255). Each time sum reaches 255, it's time to write a *1* to the speaker; otherwise the speaker gets a `0`. This has the effect of evenly distributing the ones and zeros, for a smooth output waveform. Try running this code with `DEBUG_WAVE` set to *1* to see how the ones and zeros are distributed.

the "equipment list" BIOS service. Upon returning from this call, `AX` contains bits that tell what equipment is installed. Bit *1* is supposed to be set if there is an 80x87 in the system, and cleared if not. Some clone systems don't set this bit properly. See Listing 22.3.

If your system has this problem, you can override the auto-detection code using an environment variable. Insert the statement

```
87=N
```

in your `AUTOEXEC.BAT`, or type it at the DOS prompt. If you compile your program with the `emulation` setting, this environment variable forces it to use the floating-point emulation library and not even look for an 80x87.

This behavior and the `87` environment variable are documented in Chapter 7 of the *Borland C++ Programmer's Guide* and Chapter 12 of the *Turbo C User's Guide*. Borland's other product, Turbo C++, probably does floating-point the same way, but I have not had an opportunity to use that compiler.

This bug appears only when running on 80286 and higher CPU types. On 8086/8088 processors, Borland always tests for an 8087 directly, without using the BIOS call.

Segment Lengths

If the audio file is more than a few seconds long, *TOSPKR* must pause occasionally to read in more data from disk. These annoying pauses are less obvious if you can make them occur during natural breaks in the audio (between sentences, if you're playing speech).

TOSPKR looks for a special prefix in the audio file: the character string *"LIST"*. If it exists, the program assumes it is followed by a list of audio segment lengths, each of which is an unsigned integer less than the input buffer size. *TOSPKR* reads these lengths into the `seg_length[]` array, and uses them as the number of bytes to read (`n_to_read`) during each disk read operation.

The Results

TOSPKR runs on even the slowest PC and XT systems. The number of bits per sample on these systems is very low, however, so the playback quality is not exceptional. A reasonably fast 286 or 386 produces acceptable quality for voice or sound effects.

The speaker's location and the amplifier circuit used to drive the speaker also have a great effect on the sound quality. In some PCs the speaker is buried so deep in the chassis that nothing will sound good unless you remove the system's cover.

On this book's source code disk, you'll not only get the source to all of this chapter's programs, but the executable for *TOSPKR* and some sample audio files as well. These files give you an idea of how your programs could sound if you added audio playback using these techniques.

Thanks To...

I am indebted to Art Blake, software engineer at Innovative Technology Inc., for his enthusiasm and assistance in refining the algorithms presented here.

Listing 22.1 `tospkr.c`

```
/*
 * Play an audio file file out the PC speaker.
 * Written for Turbo C 2.0, by Bob Bybee, 7/91
 *
 * usage: tospkr <filename> [<bits>]
 *
 *   <filename> is the audio file to be played.
 *
 *   <bits> is the number of speaker bits per sample.
 *          if not entered, it will be calculated.
 */
#include <stdio.h>
#include <stdlib.h>
#include <mem.h>
#include <bios.h>
#include <alloc.h>
#include <dos.h>
#include <conio.h>

#ifndef __LARGE__
#error must use LARGE memory model!
#endif

/* audio sample rate, per second */
#define SAMPRATE 8000

/*
 * 80x86 opcodes for the instruction sequences we
 * will need, in order to drive the PC speaker.
 *
 *  On entry, to set up the registers:
 *      BA 61 00    mov     dx,61h
 *      B5 4A       mov     ch,4ah
 *      B1 48       mov     cl,48h
 *
 * (Note: the 4a/48 values, at offset 4/6, should be
 * changed to agree with the current value of port 61H.)
 *
 * To set the speaker bit high:
 *      8A C5       mov     al,ch
 *      EE          out     dx,al
 *
 * To set it low:
 *      8A C1       mov     al,cl
 *      EE          out     dx,al
 *
 * A "far return" to end the routine:
 *      CB          retf
 */
```

Listing 22.1 *continued*

```
unsigned char ops_begin[] = {0xba, 0x61, 0x00, 0xb5, 0x4a, 0xb1, 0x48};
unsigned char ops_high[]  = {0x8a, 0xc5, 0xee};
unsigned char ops_low[]   = {0x8a, 0xc1, 0xee};
unsigned char ops_end[]   = {0xcb};

/* input buffer size, and pointer to it */
#define INBUFSIZE 65000U
unsigned char *inbuf;

/* array of audio segment lengths in file */
#define MAX_SEGS 200
unsigned int seg_length[MAX_SEGS];
unsigned int num_segs = 0;
char got_lengths = 0;

/* PFV is a pointer to a function taking no args, and returning void */
typedef void (*PFV)( void );
PFV playfuncs[256];

unsigned int bits_per_sample;

/* prototypes */
void patch_opcodes( void );
int best_rate( void );
int test_time( int n_bits );
void generate_codes( void );

void main( int ac, char **av )
    {
    FILE *fp;
    unsigned int nbytes, len, seg, n_to_read;
    unsigned char *p;

    --ac;
    ++av;
    if (ac < 1)
        {
        printf("usage: tospkr <voicefile> [<bits>]\n");
        exit(1);
        }

    /* open the input file */
    if ((fp = fopen(*av, "rb")) == NULL)
        {
        printf("Can't open voice file: %s\n", *av);
        exit(1);
        }
```

Listing 22.1 *continued*

```
    /* allocate memory for the input buffer */
    if ((inbuf = calloc(1, INBUFSIZE)) == NULL)
        {
        printf("Can't get memory for input buffer!\n");
        exit(1);
        }

    /* get the current value of port 61H */
    patch_opcodes();

    if (ac >= 2 && (bits_per_sample = atoi(av[1])) > 0)
        ;   /* got our sample rate on the command line */
    else
        bits_per_sample = best_rate();

    /* Create the 256 executable functions */
    generate_codes();

    /* See if the file is prefixed with a list of audio
     * segment lengths.  If so, read it into the seg_length[]
     * array.  It contains unsigned ints, zero-terminated.
     */
    fread(inbuf, 4, 1, fp);
    if (memcmp(inbuf, "LIST", 4) == 0)
        {
        got_lengths = 1;
        do  {
            fread(&len, 2, 1, fp);
            seg_length[num_segs++] = len;
            } while (len != 0);
        }
    else
        ;   /* We could rewind here, but
             * 4 bytes isn't worth it.
             */
```

Listing 22.1 *continued*

```
    /* Read in the file and play it thru the speaker.
     * On each read, get the next segment's worth (if a
     * list of segments exists), or as much as we can.
     */
    for (seg = 0; ; ++seg)
        {
        n_to_read = got_lengths ? seg_length[seg] : INBUFSIZE;
        if (n_to_read == 0 || (nbytes = fread(inbuf, 1, n_to_read, fp)) <= 0)
                break;

        p = inbuf;
        disable();

        /* For each byte, call one of 256 functions which
         * were created in the playfuncs[] array.
         */
        while (nbytes-- > 0)
            (*playfuncs[*p++])();

        enable();
        if (kbhit())    /* if a key was hit, */
            {
            getch();    /* eat the key, */
            break;      /* and quit. */
            }
        }

    fclose(fp);
    exit(0);
    }

/*
 * Read from port 61H.  Use this value with bit 1
 * set and cleared, in ops_begin[] opcodes.
 */
void patch_opcodes( void )
    {
    int val;

    val = inportb(0x61);
    val &= ~0x03;               /* lose bits 0, 1 */
    ops_begin[6] = val;
    ops_begin[4] = val | 0x02;  /* add bit 1 */
    }
```

Listing 22.1 *continued*

```
/*
 * Calculate the number of bits we can play to the
 * speaker, for each audio sample, in order to make
 * the playback come out at SAMPRATE samples per second.
 */
int best_rate( void )
    {
    int bits1, bits2, ticks1, ticks2;
    float fbits;

    /* Start with 10 bits/sample. Double it until
     * it takes at least one second (18 ticks) */
    printf("I'm timing your CPU, please wait...\n");
    bits1 = 10;
    while ((ticks1 = test_time(bits1)) < 18)
        bits1 *= 2;

    /* Now time it at 5x this number of bits. */
    bits2 = bits1 * 5;
    ticks2 = test_time(bits2);

    /* Interpolate to get the optimum number of bits
     * per sample for this PC. */
    fbits = (18.2 - ticks1) * (bits2 - bits1);
    fbits = fbits / (ticks2 - ticks1) + bits1;
    return ((int)fbits);
    }

/*
 * Build a set of instructions into inbuf, which will
 * play n_bits out to the speaker.  Return the number
 * of BIOS ticks it takes to play this many bits out,
 * SAMPRATE times.
 */
int test_time( int n_bits )
    {
    unsigned char *p;
    int i;
    long t1, t2;
    PFV testcode;

    p = inbuf;
    memcpy(p, ops_begin, sizeof(ops_begin));
    p += sizeof(ops_begin);
    for (i = 0; i < n_bits; ++i)
        {
        memcpy(p, ops_high, sizeof(ops_high));
        p += sizeof(ops_high);
        }
    memcpy(p, ops_end, sizeof(ops_end));
```

Listing 22.1 *continued*

```
    testcode = (PFV)inbuf;
    t1 = biostime(0, 0L);
    for (i = 0; i < SAMPRATE; ++i)
        (*testcode)();
    t2 = biostime(0, 0L) - t1;
    printf("bits: %d  ticks: %d\n", n_bits, (int)t2);
    return (int)t2;
    }

/*
 * Generate opcode streams, in allocated memory, to push
 * the proper bit patterns out to the PC speaker.
 */
#define DEBUG_WAVE 0    /* 1 to enable printouts */
void generate_codes( void )
    {
    unsigned char *p;
    unsigned int value, sum, i;

    printf("generating code for %d bits per sample...\n",
        bits_per_sample);
    for (value = 0; value < 256; ++value)
        {
        /* Get memory for the next instruction stream we're going to create. */
        p = calloc(1, sizeof(ops_begin) + sizeof(ops_end) +
            bits_per_sample * sizeof(ops_high));
        if (p == NULL)
            {
            printf("out of memory at value %d\n", value);
            exit(1);
            }

        playfuncs[value] = (PFV)p;
#if DEBUG_WAVE
        printf("\n%3d: ", value);
#endif
        memcpy(p, ops_begin, sizeof(ops_begin));
        p += sizeof(ops_begin);
```

Listing 22.1 *continued*

```
        /* evenly distribute the ones over the bits. */
        sum = 0;
        for (i = 0; i < bits_per_sample; ++i)
            {
            sum += value;
            if (sum >= 255)
                {
                sum -= 255;
                memcpy(p, ops_high, sizeof(ops_high));
#if DEBUG_WAVE
                putchar('1');
#endif
                }
            else
                {
                memcpy(p, ops_low, sizeof(ops_low));
#if DEBUG_WAVE
                putchar('.');
#endif
                }
            p += sizeof(ops_low);
            }
        memcpy(p, ops_end, sizeof(ops_end));
        }
    }

/* End of File */
```

Listing 22.2 u2bin.c

```
/*
 * Convert a Mu-Law encoded file to offset binary.
 * Written for Turbo C 2.0, by Bob Bybee, 7/91
 *
 * Usage: u2bin <infile> <outfile> [expon] [clip]
 *
 *   infile    is Mu-Law encoded file
 *   outfile   is offset binary file for TOSPKR
 *   expon     is exponent for uncompression (optional)
 *   clip      is clipping factor (optional)
 *
 * Defaults of expon = 3, clip = 5 seem to sound good.
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define DEFAULT_EXP  3.0
#define DEFAULT_CLIP 5.0

int xlat[128];

void main( int ac, char **av )
    {
    int c_in, c_out, sign, i;
    FILE *in_fp, *out_fp;
    float mag, expon, clip;

    /* See if <infile> and <outfile> were given, open them if so. */
    --ac, ++av;
    if (ac < 2)
        {
        printf("usage: u2bin <infile> <outfile> " "[expon] [clip]\n");
        exit(1);
        }

    if ((in_fp = fopen(*av, "rb")) == NULL || (out_fp = fopen(av[1], "wb")) == NULL)
        {
        printf("can't open files!\n");
        exit(1);
        }
```

Listing 22.2 *continued*

```
    if (ac >= 3 && (expon = atof(av[2])) > 0.0)
        ;   /* use command-line exponent */
    else
        expon = DEFAULT_EXP;

    if (ac >= 4 && (clip = atof(av[3])) > 1.0)
        ;   /* use command-line clipping */
    else
        clip = DEFAULT_CLIP;

    /* Create data translation table for these values of expon and clip. */
    for (i = 0; i <= 127; ++i)
        {
        mag = clip * 127.0 * pow((127 - i) / 127.0, expon);
        if (mag > 127.0)
            mag = 127.0;    /* clip */
        xlat[i] = (int)mag;
        }

    /* Read in the file and translate each byte. */
    while ((c_in = getc(in_fp)) != EOF)
        {
        if (c_in <= 127)
            sign = 1;
        else
            sign = -1;

        c_out = (sign * xlat[c_in & 127]) + 127;
        putc(c_out, out_fp);
        }

    fclose(in_fp);
    fclose(out_fp);
    exit(0);
    }

/* End of File */
```

Listing 22.3 `int11.c`

```
/*
 * Print BIOS equipment list from "int 0x11".
 * Written for Turbo C 2.0, by Bob Bybee, 9/91.
 *
 * Bit 1 is set if the BIOS thinks there's an 80x87
 * installed, clear if not.  See any BIOS book for
 * detailed descriptions of the other bits.  Not all
 * BIOS versions use these bits in the same ways.
 */
#include <stdio.h>
#include <dos.h>

void main( void )
    {
    union REGS regs;
    unsigned int ef;    /* equipment flags */

    int86(0x11, &regs, &regs);     /* ask BIOS for it */
    ef = regs.x.ax;                /* copy the AX value */

    printf("Selected BIOS equipment flags from INT 0x11:\n");
    printf("    math coprocessor: %u\n", (ef >> 1) & 1);
    printf("  initial video mode: %u\n", (ef >> 4) & 3);
    printf("# of diskette_drives: %u\n", ((ef >> 6) & 3) + 1);
    printf("      # of com ports: %u\n", (ef >> 9) & 7);
    printf("      # of lpt ports: %u\n", (ef >> 14) & 3);
    }

/* End of File */
```

Chapter 23

Event Timing on MS-DOS PCs

Phyllis K. Lang

An MS-DOS PC system may be viewed as a layered system (Figure 23.1), consisting of MS-DOS at the top, the BIOS underneath and the PC hardware on the bottom. Each of these three layers harbors its own solution to event timing. The precision required by your application determines which of the three timing alternatives should be used.

The time of day service provided by MS-DOS may be used for timing events with duration on the order of seconds or more. This service is available through MS-DOS interrupt `21H`, function `2CH`. The precision obtained is approximately 55ms, but the time returned by MS-DOS is formatted in hours, minutes, seconds, and hundredths of seconds. This format requires special consideration at midnight.

The same 55ms accuracy is available to applications via a BIOS interrupt whose service may be customized by manipulating the MS-DOS interrupt vector table and writing a corresponding interrupt service routine. It is imperative that interrupt service routines be written efficiently so that system performance is not degraded. Furthermore, the system environment must be maintained by the interrupt service.

The MS-DOS timing chain includes a free BIOS interrupt (`1CH`) for applications. This interrupt is issued every 55ms by the system time tick interrupt (`int 8`). The standard MS-DOS interrupt `1C` handler does essentially nothing (an `iret` instruction). The standard handler may be replaced by a custom handler which might, for example, decrement a globally defined counter that may be interrogated by other procedures. The custom handler should be installed by using MS-DOS interrupt `21`, functions `25` and `35` to change the interrupt vector. This is the safe method of changing interrupt vectors and should be used whenever possible.

Since interrupt `1C` is a free vector for applications, beware of resident programs that use it. At a minimum, you should restore the original vector at program exit. To minimize conflicts with other applications, the new interrupt `1C` handler should chain to any existing handler as in Listing 23.1. This listing illustrates how to write the interrupt handler, how to install it, and how to use the new timing service. Both Listing 23.1 and Listing 23.2 are written in Turbo Pascal v5.0 which provides the necessary interrupt service support. The stack is adjusted, the system flags are saved and the necessary `iret` instruction is generated by the `interrupt` keyword in the procedure header. Inline assembly language is used in the procedure `sti` to enable interrupts and procedure `intchain` to daisy chain interrupts.

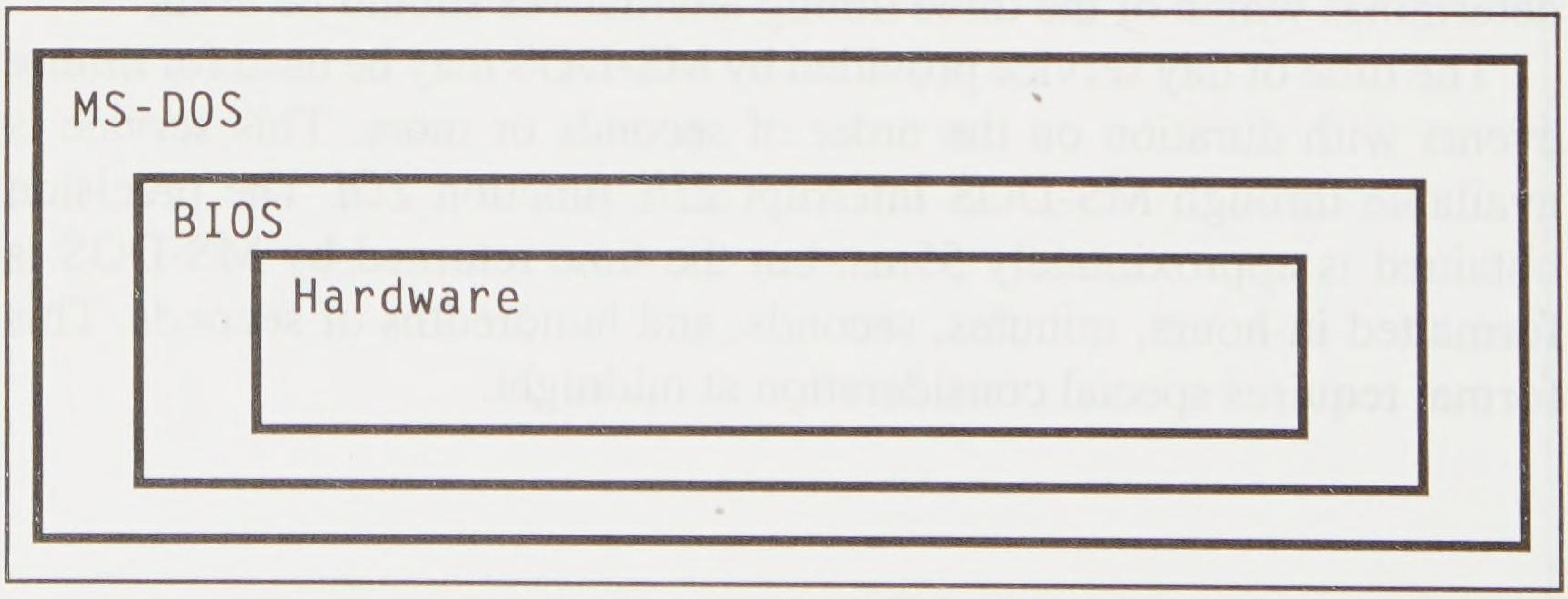


Figure 23.1 ***An MS-DOS PC system viewed as a layered system.***

Some applications may demand more accuracy than `int 1C`'s 55ms resolution. The MS-DOS PC time chain is designed to provide 55ms accuracy; therefore, to increase this precision you must change the operation of both MS-DOS and the PC hardware.

Figure 23.2 shows the PC timing chain. A system clock oscillates at 14.318MHz. This pulse rate is divided by various components to 1.19318MHz, which becomes the input frequency to the i8253 Programmable Interval Timer (PIT). The PIT contains three independently programmable timers, each of which may be loaded with a different 16-bit down count value. Each timer also may be programmed to operate in one of six modes. The mode selection dictates how the timer behaves upon decrementing to zero. MS-DOS programs `timer 0` to output a square wave at the terminal down count of 65,536. The output of `timer 0` is tied directly to an i8259 Programmable Interrupt Controller (PIC). Thus `timer 0` produces a hardware interrupt every 54.9ms. (1.19318MHz / 65536 = 18.17874Hz; 1 / 18.17874Hz = 54.9ms). The PIC is programmed to map the time interrupt to `int 8` in the interrupt vector table. This is the "hook" into the timer interrupt system.

Timing resolution may be improved by increasing the frequency of timer interrupts by loading `timer 0` with a smaller down count value. However, the MS-DOS `int 8` handler controls many system resources which rely upon the 54.9ms heartbeat. If the `int 8` interrupt frequency is increased, other system functions are serviced more frequently as well. The challenge is to ensure that MS-DOS functions are serviced at the normal rate of 54.9ms while the timer interrupt occurs more frequently.

The program in Listing 23.2 accomplishes this by saving the MS-DOS `int 8` vector, replacing it with a vector to the new handler, and chaining to the normal MS-DOS `int 8` handler at the proper time. The interrupt rate is increased by an integral factor so that the MS-DOS service may be called at a regular frequency. `Timer 0` is programmed in `i8swapin` to

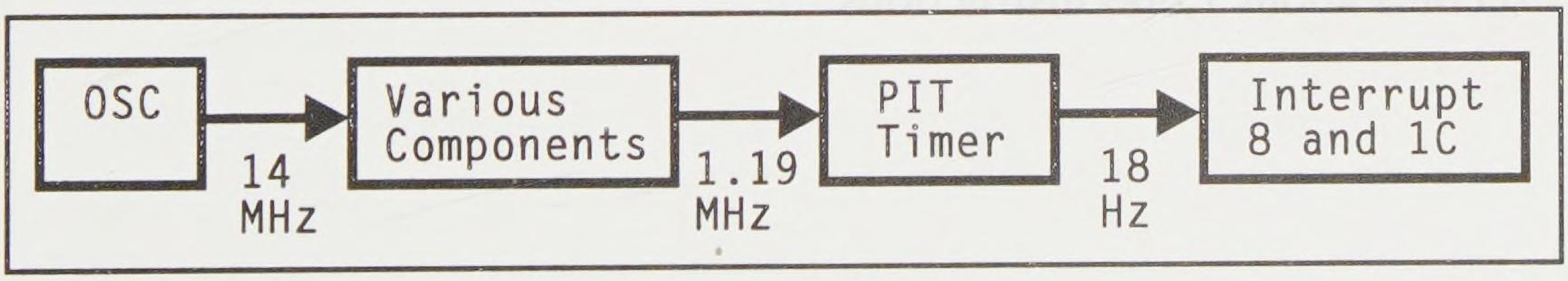


Figure 23.2 ***The PC timing chain.***

operate at a frequency of 16 times the normal rate, or one interrupt every 3.43ms. The new handler, *i08hndlr*, chains to the MS-DOS *int 8* handler every sixteenth call.

A final issue involves installing the new timing scheme. PIT *timer 0* must be reprogrammed to interrupt at the new rate on an even boundary of the 54.9ms interrupt; otherwise, some time in the system is lost and the system clock will show it. Likewise, the old rate must be restored in a similar fashion. Two additional *int 8* interrupt handlers synchronize the changes. *i8swapin()* in Listing 23.2 is called on the 54.9ms interrupt. It programs PIT *timer 0* to operate at the increased frequency and installs the new interrupt handler, *i08hndlr()*. Before exiting the program, *i8swapout()* must be called to restore the timer interrupt vector. MS-DOS restores some interrupt vectors itself at program exit; the timer interrupt vector is *not* among them, so it must be restored by the application. *i8swapout()* is called on the 3.43ms interrupt. At the boundary of the normal 54.9ms interrupt, it programs PIT *timer 0* to operate at the normal frequency and restores the MS-DOS *int 8* vector. Restoring on the interrupt boundary assures that no system time is lost.

The interrupt frequency can be altered by changing *int8_passcount_reset* in Listing 23.2. Realize, though, that there is an upper limit to the interrupt frequency. The service of an interrupt occupies some of the execution time available before the next interrupt occurs. If the interrupt is programmed to occur every 100 µs and the service requires 100 µs to execute, no time remains for the application: the system locks up! Of course the processor clock speed also must be considered in determining the upper limit of the interrupt frequency. A 4.77MHz i8086 operates at about 210ns per cycle. Assuming the average i8086 instruction requires seven cycles, one instruction executes in 1.47 µs. Thus, the 3.43ms interrupt frequency described above allows about 2,300 instructions to execute between interrupts — not enough time for a significant amount of overhead. In the same interval a 16MHz i80386 executes about 7,800 instructions.

Summary

Event timing may be accomplished with MS-DOS time of day service, BIOS interrupt *1C*, or reprogramming PC hardware. A problem may arise in using the BIOS interrupt if a terminate-and-stay-resident program tries to use that vector after it has been changed. Reprogramming the hardware may be dangerous if the system hardware differs from that described or if the application does not restore interrupt vectors properly. In addition, there may be several exit routes in a program: an application defined route, *Ctrl-Break* (see Chapter 11) and the critical error handler's (see Chapter 10) abort option. Each of these possibilities should be addressed. If these problems are properly addressed, the timing resolution attainable with an MS-DOS PC can accommodate many demanding real-time applications.

Listing 23.1 I1CEX

```
{ --------------------------------------------------------------------------
  I1CEX inserts a user defined timer tick interrupt service procedure
  into the MS-DOS timekeeping chain.

  Author: pkl
  -------------------------------------------------------------------------- }

{$F+}

  program i1cexample;

  uses
    dos, crt;

  var
    countdowntimer : integer;
    int1cvec : pointer;

{ --------------------------------------------------------------------------
  STI enables interrupts.
  -------------------------------------------------------------------------- }

  procedure sti; inline($fa);

{ --------------------------------------------------------------------------
  INTCHAIN chains to an interrupt.  The stack is prepared so that the DOS
  interrupt handler will receive it in proper condition.
  -------------------------------------------------------------------------- }

  procedure intchain(ivec : pointer);

  begin
    inline($9c/                           { pushf }
           $ff/$5e/$06);                  { call dword ptr [bp+6] }
  end;
```

Listing 23.1 *continued*

```
{ -------------------------------------------------------------------------
  I1CHNDLR is an interrupt handler for the user timer tick (1C) interrupt.

  The BIOS time tick interrupt (int 8) service routine (TIMER_INT) updates
  the time of day clock, checks the diskette motor counter for timeout, and
  issues a user time tick interrupt (int 1C). This routine decrements a count-
  down timer maintained in external memory where it may be set and polled by
  functions within the program.
  ------------------------------------------------------------------------- }

  procedure i1chndlr(flags,cs,ip,ax,bx,cx,dx,si,di,ds,es,bp : word);
  interrupt;

  begin

    sti;                                 { enable interrupts }

    countdowntimer := countdowntimer - 1;
    intchain(int1cvec);                  { chain to the preexisting handler }
  end;

{ -------------------------------------------------------------------------
  Insert a 54.9 ms countdown timer into the MS-DOS timing chain that
  is accessible to the application.
  ------------------------------------------------------------------------- }

  begin

    GetIntVec($1c, int1cvec);            { Save present vector }
    SetIntVec($1c, @i1chndlr);           { Install new one }

{ Example: delay for 200 ms. }

    countdowntimer := 200 div 55;
    while countdowntimer > 0 do;

{ Example: wait for a keyboard stroke for 2 seconds. }

    countdowntimer := 2000 div 55;
    write('Press any key>');
    while (countdowntimer > 0) and (not KeyPressed) do;

    if (countdowntimer <= 0)    then    { Time out on keyboard }
      writeln(' ** Keyboard timed out **')
    else
      writeln(' ** Ok **');

    SetIntVec($1c, int1cvec);           { Restore the interrupt 1C vector. }

  end.

{ End of File }
```

Listing 23.2 I8EX

```
{ ---------------------------------------------------------------------------
  I8EX reprograms the i8253 PIT to interrupt at 16 times it normal rate
  of 54.9 ms.  The interrupt decrements a user accessible timer every
  3.43 ms.

  Author: pkl.
  --------------------------------------------------------------------------- }

{$F+}

  program i8ex;

  uses
    dos, crt;

   type
    itab = array [0..$ff] of pointer;

  var
    int08vec : pointer;                 { Interrupt 08 save vector }
    int8_passcount : integer;           { Current interrupt downcounter }
    timer_count : integer;              { Application countdown clock timer }
    vectab : itab absolute $0:$0;       { 80x86 interrupt table }

  const
    isr_8259 : word = $0020;            { i8259 in-service reg addr }
    eoi_8259 : byte = $20;              { i8259 end-of-intrpt instr }

    cwr_8253 : byte = $43;              { control word reg addr }
    ctr0_8253 : byte = $40;             { timer 0 counter addr }
    set_ctr0_8253 : byte = $36;         { Binary mode 3 ctr 0 }

{ Define speedup factor and the corresponding i8253 counter values.
  The normal timer 0 interrupt occurs every 54.9 ms. The frequency of that
  interrupt is increased by a factor of 16 for this application so
  interrupts occur every 3.43 ms.                                          }

    int8_passcount_reset : integer = 16;

{ The input clock frequency is 1.19318 MHz on the PC; the corresponding
  downcount values for 54.9ms and 3.43ms follow.                           }

    int8_newct_msb : byte = $10;
    int8_newct_lsb : byte = $00;
    int8_dosct_msb : byte = $00;
    int8_dosct_lsb : byte = $00;
```

Listing 23.2 *continued*

```
{ --------------------------------------------------------------------------
  STI enables interrupts.
  -------------------------------------------------------------------------- }

  procedure sti; inline($fa);

{ --------------------------------------------------------------------------
  INTCHAIN chains to an interrupt. The stack is prepared so that the DOS
  interrupt handler will receive it in proper condition.
  -------------------------------------------------------------------------- }

  procedure intchain(ivec : pointer);

  begin
    inline($9c/                         { pushf }
           $ff/$5e/$06);                { call dword ptr [bp+6] }
  end;

{ --------------------------------------------------------------------------
  I8HNDLR is an interrupt handler for the DOS time tick interrupt (int 8).
  This handler processes more frequent hardware clock interrupts while
  maintaining the normal 54.9 ms system clock updating.
  -------------------------------------------------------------------------- }

  procedure i08hndlr(flags,cs,ip,ax,bx,cx,dx,si,di,ds,es,bp : word);
  interrupt;

  begin

    sti;                               { Reenable interupts }
    timer_count := timer_count - 1;    { Decrement users timer }

{ If it is time to do the system time chores, reset the pass counter and
  chain to the normal DOS interrupt vector.
  The normal DOS handler does an sti and reenables the i8259.          }

    int8_passcount := int8_passcount - 1;
    if (int8_passcount = 0) then
    begin
      int8_passcount := int8_passcount_reset;
      intchain(int08vec);
    end
    else
      port[isr_8259] := eoi_8259;
  end;
```

Listing 23.2 *continued*

```
{ ------------------------------------------------------------------------
  I8SWAPIN is an interrupt handler for the DOS time tick interrupt (int 8).
  This handler swaps in a new handler for more frequent hardware interrupts
  while maintaining the normal 54.9 ms system clock updating. The swap
  occurs at interrupt time so that no system clock ticks are lost.
  ----------------------------------------------------------------------- }

  procedure i8swapin(flags,cs,ip,ax,bx,cx,dx,si,di,ds,es,bp : word);
  interrupt;

{ i8259 interrupts are disabled upon entry; re-enable interrupts.
  Configure the i8253 to run in mode 3 (square wave) with a new 16 bit
  binary countdown value.                                                 }

  begin

    sti;                                  { Enable interrupts }

    port[cwr_8253] := set_ctr0_8253;      { Timer output is square wave }
    port[ctr0_8253] := int8_newct_lsb;    { New down count value }
    port[ctr0_8253] := int8_newct_msb;

{ Initialize pass counter for int 8 handler.                              }

    int8_passcount := int8_passcount_reset;

{ Swap in the new time tick interrupt handler vector. DOS interrupts
  shouldn't be used within an interrupt handler.                          }

    vectab[8] := @i08hndlr;

{ Go perform DOS time services. DOS does the eoi for the i8259.           }

    intchain(int08vec);

  end;
```

Listing 23.2 *continued*

```
{ -----------------------------------------------------------------------
  I8SWAPOUT is an interrupt handler for the DOS time tick interrupt
  (int 8).  This handler swaps out the new handler and restores the DOS
  handler.  The swap occurs at interrupt time so that no system clock
  ticks are lost.
  -----------------------------------------------------------------------}

  procedure i8swapout(flags,cs,ip,ax,bx,cx,dx,si,di,ds,es,bp : word);
  interrupt;

{ i8259 interrupts are disabled at entry; re-enable.  Restore the normal
  DOS operation of i8253 timer 0: mode 3 with a binary downcounter.      }

  begin

    sti;                                { Enable interrupts }

    timer_count := timer_count - 1;     { Decrement user timer }
    int8_passcount := int8_passcount - 1;

    if (int8_passcount = 0) then
    begin                               { Time to change vectors }

      port[cwr_8253] := set_ctr0_8253; { Binary downcount mode }
      port[ctr0_8253] := int8_dosct_lsb;        { Timer expires in 54.9 ms }
      port[ctr0_8253] := int8_dosct_msb;
      vectab[8] := int08vec;            { Restore vector for interrupt 8 }
      intchain(int08vec);               { Do the DOS time service. }
    end
    else
      port[isr_8259] := eoi_8259;       { End of interrupt for i8259. }

  end;
```

Listing 23.2 *continued*

```
  var
    reply : byte;

  begin
    GetIntVec($08, int08vec);               { Save present int 8 vector }
    SetIntVec($08, @i8swapin);              { Install vector to swap in new
                                              timing scheme }

{  Example: wait 300 ms for pin 2 on LPT1: to go high (1).                 }

    timer_count := 300 * int8_passcount_reset div 55;
    reply := $00;
    while (timer_count > 0) and (reply = $00) do
    begin
      reply := port[$03BC];                 { Read the port }
      reply := reply and $01;               { Mask the desired bit }
    end;

    if (timer_count <= 0) then
       writeln('Pin 2 on LPT1: did not go high.');

    timer_count := 2000 * int8_passcount_reset div 55;
    write('Press any key>');
    while (timer_count > 0) and (not KeyPressed) do;

    if (timer_count <= 0)       then    { Time out on keyboard }
      writeln(' ** Keyboard timed out **')
    else
      writeln(' ** Ok **');

{ Return the system to its normal timekeeping.  Install a vector to the
  handler that reprograms the PIT and restores the DOS handler at the next
  interrupt.                                                               }

    SetIntVec($08, @i8swapout);
    writeln('Give vector time to swap>');
    while (not KeyPressed) do;
  end.

{ End of File }
```

Chapter 24

Writing Device Drivers with Turbo C

Robert Allen

I read with interest the article "Writing MS-DOS Device Drivers" by Philip J. Erdelsky in *The C Users Journal* vol 6., no. 7. In the last few years I have written three different device drivers for MS-DOS, each to perform fully buffered serial communications. The first driver was written in assembly, the second was attempted in Lattice C v3.0, but it became so cumbersome I reverted to assembly. Finally, on my third attempt I found a compiler, Borland's Turbo-C, that makes the writing of device drivers in C very simple.

My tools are Turbo-C v1.0 and Microsoft MASM v5.0 which includes the MAKE and Linker. My editor is VEDIT by CompuView. My driver uses 8Kb of memory, but offers a 3Kb receive buffer, sophisticated IOCTL capabilities, hums along at 9600 baud, and even has an on-screen, switchable, debugging mode. Except for the single startup file written in assembly, the "hard" stuff is all written in C-45 source modules representing over 3,000 lines of C source code.

Turbo-C offers the following advantages specific to device drivers:

- Support for the `tiny` programming model. In this model registers *CS*, *DS*, *ES*, and *SS* all have the same value. This is crucial for writing small and efficient device drivers. The standard MS-DOS utility *EXE2BIN* will convert an `.EXE` binary file compiled in `tiny` model to the `.SYS` driver format.
- The ability to control the data segment name and group through command line options. (Appendix C of the "Reference Guide" offers a good explanation of the wide range of options.)
- The ability to mix `far` pointers and functions within the `tiny` model. This is critical as the MS-DOS "Request Header" is a `far` pointer and the strategy and interrupt routines must do `far` returns upon exit. The rest of the driver need only use short addresses. A header file called `rqh.h` has been included to demonstrate the mixing of pointer types.
- In-line assembly may be included with the `#pragma` directive, a powerful method for switching to assembly to perform specialized instructions — all in the middle of C code.
- Direct access to machine registers with pseudo-variables (page 264 of the *User's Guide*). This direct access coupled with in-line function calls for I/O bus operations make for efficient code.
- The object file format is compatible with the Microsoft linker, librarian, and assembler as well as advanced development tools such as Phoenix's PLINK and PFIX.
- The Turbo Language extension, "functions of type `interrupt`," allows the programmer to specify that a function should save all registers on entrance and perform an *IRET* instruction on exit.

An Example

With Turbo C the two trickiest steps in developing a driver are writing the startup code and correctly linking the pieces.

The Startup Code

The 58 lines of Listing 24.1 are the only assembly language in my driver. This file contains the header structure for the driver and declares all the segments and groups used. Declaring the segments in the proper order provides the linker with the needed instructions for organizing the driver.

A few concessions to Turbo-C are required to enable the use of the standard library, but they are trivial and the standard library has many useful utilities.

Linking

A device driver requires the header structure at the very start of the file. This is unquestionably data. But most compilers (and, in fact Turbo-C) demand the code be first, followed by initialized data and the stack. Even in the `tiny` model the compiler still wants to make data addressing relative to the end of the code.

My driver uses memory as illustrated in Figure 24.1. This structure can be created with careful use of command line arguments to the TCC stand-alone compiler when building the `.OBJ` modules. The right options will put the driver initialization code at the end of the code segment, and all the initialization data in a segment of its own at the end of the driver.

I put all my initialization modules in a separate library to be included at link time. The actual routine called to service the MS-DOS `Command 0` is called `df_init()`. My routine (as well as all run-time routines) is compiled with the `make` commands:

```
CFLAGS = -I<standard includes> -mt -f- -K -Z -c
TCC $(CFLAGS) $**
```

```
----------
Device Driver Header Structure (Data)
----------
Run Time Device Driver Code
---------- (start of receive buffer)
Driver Initialization-only Code
----------
Turbo-C Standard Library Routines
---------- (end of receive buffer)
Run Time Device Driver Data
---------- (start of transmit buffer)
Driver Initialization-only Data
----------
```

Figure 24.1 ***Device driver memory structure example.***

The *CFLAGS* specify the *tiny* model, no floating point, *char* is type *unsigned*, some optimization, and build *.OBJ*s only.

In addition, the *df_init()* routine was added to a special library with these *make* statements:

```
TAIL =the_tail
LIB $(TAIL) -+$*;
```

In my driver the *df_init()* routine not only performs the *UART* interrupt initialization and buffer allocation, but also generates extensive messages describing the driver's configuration. Also, I make extensive use of the command line in the *CONFIG.SYS* file — *df_init()* handles the required parsing.

All of this work was broken into modules where separate routines handle logical pieces of the task. The routines that are called only by *df_init()* are compiled with TCC command line arguments which force the data space used by these routines into a separate segment and group. The *make* commands are:

```
TCC -zR_TAIL -zTTAIL $(CFLAGS) $**
LIB $(TAIL) -+$*;
```

The two *-z* commands relate to a special segment declared in the setup file *DVR_MAIN.ASM*.

The only caveat to this approach is that the routines called by *df_init()* cannot access the normal global data variables used by the run-time routines (which are oriented to the normal run-time segment *_TEXT* and not to the new initialization segment *_TAIL*). The linker will not be able to resolve references to these globals because it expects data references to be relative to the data segment, not to the start of the file.

The solution is to have *df_init()* pass pointers to the data as parameters on the stack. These pointers have long since been resolved by the linker, so the called function receives accurate addresses.

In my driver the initialization code takes up a "hole" in the middle of the driver. The initialization data is tagging along at the end of the driver, where it is easily discarded.

Once initialization is complete, the initialization code can be discarded and the resulting "hole" in the middle of the driver reused. I enlisted the space as the receive side ring buffer for serial communications. As the initialization code begins with the address of `df_init()` and is completed with the end of the code segment (and thus the start of the run-time data segment), it is easy to calculate the addresses.

I am careful to organize the `MAKE` file so that a known variable name is at the beginning of the data segment. In my case, it is the character pointer to the receive side ring buffer. Thus, ring buffer addresses and sizes, as well as the end of the driver, are computed with code that looks like Listing 24.2.

Essentially, control structures are initialized to have the receive buffer overlay the initialization code. The transmit buffer begins at the start of the `_TAIL` segment (which is throw-away data space after initialization). Finally MS-DOS is told that the driver ends immediately after the transmit side ring buffer.

Other Issues

The MS-DOS strategy and interrupt routines are easily implemented in Turbo-C. My header and the routines, `strategy()` and `dosio()`, are given in Listings 24.3 and 24.4. You must switch to a local stack when `dosio()` is called.

When the driver is initializing, it is allowed to do many legal things to the header structure prior to returning control to MS-DOS. My driver uses the command line to judge if it should be addressing the first or second serial port. Then appropriate changes for the device name in the header structure, to `COM1` or `COM2`, will result. Thus, to support both serial ports, the driver is merely loaded twice and initializes itself to different names.

Device drivers, during initialization, are free to use the lower-numbered MS-DOS calls to read and write to the console. This coincides nicely with the `getch()`, `putch()`, and `cputs()` function calls in the Turbo-C library. However, the programmer should be very careful where the linker puts the library routines. Availability of "normal" routines which don't need MS-DOS, `atoi()` and the string processing routines, is also very useful.

The programmer is responsible for switching to a local stack for driver processing. If anything slightly sophisticated is performed the driver will run out of stack space. With a little help from in-line assembly I use two stacks, one during MS-DOS access requests and one during interrupt processing. I even managed to write one implemented as a function call!

Conclusion

I strongly recommend the use of Turbo-C as the compiler of choice for writing MS-DOS device drivers. After years of supporting cantankerous assembly files, it is a pleasure to support an MS-DOS driver with sophisticated features written in C.

Listing 24.1 *An example header structure for a device driver.*

```
; Copyright 1987, 1988 PARS Service Partnership -- R.D.Allen, 9/4/87

;Declare Segments & Groups. Define Device Driver Header
DGROUP  GROUP   _TEXT, _DATA, _CVTSEG, _SCNSEG, _BSS, _TAIL

ASSUME  CS:_TEXT, DS:NOTHING
_TEXT   SEGMENT BYTE PUBLIC 'CODE'      ; All code goes in this segment

    PUBLIC  _header
    PUBLIC  DGROUP@
    PUBLIC  _abort
    PUBLIC  _setDGROUP

    EXTRN   _strategy:NEAR
    EXTRN   _dosio:NEAR

; DEVICE HEADER: Must be first thing in the binary file. Always!

_header DW      -1                      ; Chain Pointer -1, only device in driver
        DW      -1
        DW      1100100000000000b       ; IOCTL word, simple character device
        DW      DGROUP:_strategy
        DW      DGROUP:_dosio
        DB      "COMX    "              ; Name of driver is COMX

DGROUP@         dw ?                    ; Turbo-C library requires this set to DS

_setDGROUP:
        MOV         CS:DGROUP@, DS      ; TINY model library wants this
                                        ; Called during (df_) INIT. Falls into ret

_abort: RET                             ; Hope this is never called!

_TEXT   ENDS
_DATA   SEGMENT BYTE PUBLIC 'DATA'      ; Main data segment for driver
_DATA   ENDS
                                         ; Additional segments desired by the compilier.

_CVTSEG SEGMENT WORD PUBLIC 'DATA'
    PUBLIC  __RealCvtVector
__RealCvtVector dw ?
_CVTSEG ENDS

_SCNSEG SEGMENT WORD PUBLIC 'DATA'
_SCNSEG ENDS

_BSS    SEGMENT WORD PUBLIC 'BSS'
_BSS    ENDS

_TAIL   SEGMENT WORD PUBLIC 'TAIL'              ; Data area for initialization code
_TAIL   ENDS                                    ; All can be thrown away

        END
```

Listing 24.2 *Example code from a MAKE file.*

```
    void init_buffers(ptr, rbuf, tbuf )
    union _ptr *ptr;                /* Maps the MS-DOS request header */
    struct _ring *rbuf;             /* Maintenance of RCV ring buffer */
    struct _ring *tbuf;             /* Maintenance of XMT ring buffer */
    {
    register unsigned int i;
    extern char tail;               /* First variable in _TAIL segment*/

    i = (unsigned int) &df_init;
                                    /* Compute Receive Buffer */
    rbuf->start = (char *) i;
    rbuf->size  = (unsigned int) rbuf - i;

                                   /* Compute Transmit Buffer */
    tbuf->start = &tail;
    tbuf->size  = TRANSMIT_SIZE;

                                   /* Compute end of driver */
    ptr->rq->tail.init.end.address.segment = _CS;
    ptr->rq->tail.init.end.address.offset  = (unsigned int) &tail + TRANSMIT_SIZE;
    }

/* End of File */
```

Listing 24.3 RQH.H

```
/* RQH.H **     Device Driver definitions and templates.
         **     The DOS device header and DOS Request Header is mapped.
                                                    -- R.D.Allen, 9/3/87
Copyright 1987, 1988 PARS Service Partnership
*/

/* Convenient method for mapping an INTEL far pointer in C */

union _farp {
    struct {
        unsigned int offset;
        unsigned int segment;
    } address;
    char far *string;
};
```

Listing 24.3 *continued*

```
/* A method to map the device header structure required by DOS
   and declared in the startup file DVR_MAIN.ASM                */

extern struct _header {
    union _farp next_dev;
    union _ioctl {
        unsigned int word;
        struct {
            unsigned int stout      : 1;    /* Bit 0 ** Is stdout device      */
            unsigned int stin       : 1;    /* Bit 1 ** Is stdin device       */
            unsigned int nul    : 1;        /* Bit 2 ** Is the NULL device    */
            unsigned int clock       : 1;   /* Bit 3 ** Is the $CLOCK device */
            unsigned int rsv4       : 1;    /* Bit 4 **     reserved          */
            unsigned int raw    : 1;        /* Bit 5 ** Raw mode status       */
            unsigned int eof    : 1;        /* Bit 6 ** At end of file status */
            unsigned int isdev      : 1;    /* Bit 7 ** Is a device           */
            unsigned int rsv810     : 3;    /* Bits 8 - 10 ** reserved        */
            unsigned int ocr    : 1;        /* Bit 11 ** I forgot?            */
            unsigned int rsv12      : 1;    /* Bit 12 ** reserved             */
            unsigned int obusy      : 1;    /* Bit 13 ** Output busy status   */
            unsigned int ioctl      : 1;    /* Bit 14 ** Supports IOCTL call  */
            unsigned int cdev       : 1;    /* Bit 15 ** Character device     */
        } bits;
    } io;
    unsigned int strat;                     /* Address of STRATEGY Routine   */
    unsigned int intr;                      /* Address of INTERRUPT Routine  */
    char name[8];                           /* DOS file name of device       */
} header;

/* The tail of a Request Header is mapped differently for different
   commands. These structures will define all necessary access methods. */

struct _init {                /* Command tail used only during initialization    */
    char    units;            /* Number of Units as required for Block Device    */
    union   _farp end;        /* Ending address of driver reported to DOS        */
    union   _farp args;       /* CONFIG.SYS line after "DEVICE="                 */
    char    devnum;           /* Beginning Device Number, this Block Device      */
};

struct _io {                  /* Command tail used for most other commands       */
    char    media;            /* Block Device media field                        */
    union   _farp data;       /* Address for data Input/Output                   */
    int         count;        /* Number of bytes to be transferred               */
    int         sec_num;      /* Block Device sector number                      */
    union   _farp volume;     /* Block Device Volume label                       */
};
```

Listing 24.3 *continued*

```
/* A union to hold all the different Request Header command tail formats.     */

union _rq_tail {
    struct _init    init;     /* Initialization command tail                  */
    struct _io          io;   /* Most read, write routines                    */
    char            peek;     /* Non-destructive read, no wait                */
};

/* DOS Request Header format. This structure unites all the different
   parts of a request header so they may be accessed in C.                    */

struct _rq_hdr {
    char    count;                      /* total bytes in header               */
    char    unit_code;                  /* Block Device Garbage (not used)     */
    char    command;                    /* Device driver command               */
    struct  _st_bits {                  /* Bit fields to map Status word       */
        unsigned int err_type   : 8;    /** DOS error to report                */
        unsigned int done       : 1;    /** Done status (normally returned)    */
        unsigned int busy       : 1;    /** Busy status (normal not used)     */
        unsigned int reserved   : 5;
        unsigned int err_flag   : 1;    /** If set, ERR_TYPE should used       */
    } status;                           /* Status word returned to DOS         */
    struct  _rq_hdr far *next_rq;       /* RESERVED -- front pointer           */
    struct  _rq_hdr far *pending;       /* Chain pointer for queued req.       */
    union   _rq_tail tail;              /* Request Header Command Tail         */
};

/* Request Header address storage as used by STRATEGY routine. */

union _ptr {
    struct {
        unsigned int offset;
        unsigned int segment;
    } address;
    struct _rq_hdr far *rq;         /* far pointer to access request header */
};

/* End of File */
```

Listing 24.4 DOSIO.C

```
/* DOSIO.C **   MS-DOS Input/Output routines for Character Device Driver.
                                                        -- R.D.Allen, 9/4/87

Copyright 1987, 1988, PARS Service Partnership
*/

#include    <stdio.h>
#include    "driver.h"    /* Function prototypes */
#include    "rqh.h"       /* Request header structure */

#define DF_MAX      17
                          /* Jump Table used only here to route DF command proccessing.
*/
int         (*jmp_table[DF_MAX])() = {
    df_init,              /* CMD 0 -- Initialization                          */
    df_noop,              /* CMD 1 --     Media Check (Block Only)            */
    df_noop,              /* CMD 2 -- Build BPB (Block Only)                  */
    df_ioctl,             /* CMD 3 -- I/O Control Input (Read from DOS)       */
    df_read,              /* CMD 4 --     Normal Device Read                  */
    df_peek,              /* CMD 5 --     Non-destructive input, no wait      */
    df_rstat,             /* CMD 6 -- Read Status                             */
    df_rflush,            /* CMD 7 -- Read Flush                              */
    df_write,             /* CMD 8 --     Normal Device Write                 */
    df_write,             /* CMD 9 -- Write with verify (Same as write, here) */
    df_wstat,             /* CMD A -- Write Status                            */
    df_wflush,            /* CMD B -- Write Buffer Flush                      */
    df_ioctl,             /* CMD C --     I/O Control Output (Write to DOS)   */
    df_open,              /* CMD D --     Device Open (new for 3.0)           */
    df_close,             /* CMD E -- Device Close (new for 3.0)              */
    df_noop,              /* CMD F --     Removable Media (Block Only)        */
    df_write              /* CMD 10 -- Write until busy (Same as write, here) */
};
```

Listing 24.4 *continued*

```
/*********
*    NAME:     DOSIO      (a far function)
*     USE:     This is called directly by DOS to parse a command header.
*              Use the command number passed and call the appropriate
*              processor. The return value is the error flag.
*   INPUT:     void (use global Request Header pointer from STRATEGY)
*  OUTPUT:     void (but a non-zero result from jump table list is an
*                error value and the error status bit is set)
*********/

void far dosio(void)
{
static int result;          /* Must be static to live after stack_new          */

    stack_new();            /* Switch to local stack for this module           */

    result = (*jmp_table[ptr.rq->command])(ptr.rq->command);

    if(result) {
        ptr.rq->status.err_type = result;
        ptr.rq->status.err_flag = 1;
    }
    ptr.rq->status.done = 1;

    stack_old();            /* Switch back to callers stack                    */
}
```

Listing 24.4 *continued*

```
/* STRATEGY.C ** "Strategy" code as defined for device drivers.
                                                      -- R.D.Allen, 9/4/87
Copyright 1987, 1988 PARS Service Partnership
*/

#pragma inline

#include    "rqh.h"

/*********
*    NAME:     STRATEGY       (a far function)
*     USE:     Store the address of the request header for later
*              processing by DOSIO. This is the classic "strategy" routine
*              as described in Device Driver manuals. DOS passes the request
*              header and this routine saves it for DOSIO (the "interrupt"
*              routine) to parse.
*              Can be done in C with pseudo variables (but ASM is tighter):
*                  ptr.address.offset = _BX;
*                  ptr.address.segment = _ES;
*   INPUT:     void
*  OUTPUT:     void
*********/

void far strategy(void)
{
asm         extrn _ptr:near
asm         mov     cs:Word Ptr _ptr,          bx
asm         mov cs:Word Ptr _ptr + 2,      es
}

/* End of File */
```

Chapter 25

Using CD-ROM Device Drivers

Dirk S. Baum

CD-ROM discs provide a convenient method for storing large amounts of prerecorded material. The availability of CD-ROM drives for the PC has made it practical to distribute very large data bases to many people. CD discs are most widely used for the storage of digital audio information. In fact, the use of CD-ROM for storing data is a direct outgrowth of the use of this medium by the music industry. This chapter will concentrate on how to control the playing of CD audio discs in a CD-ROM drive attached to your PC.

There are several reasons for playing CD audio discs with your computer. You might wish to play your favorite CDs while working at your computer. Besides using your CD-ROM drive for this type of pleasure or relaxation, you can integrate audio from CDs with your applications. It is possible to synchronize the audio with the actions of your application. One of the nice features of CD-ROM drives allows you to play audio without constant program intervention. Once you start the playing, you do not have to do anything else. The audio will continue to play until you explicitly stop it or until it reaches the stopping point specified when you started the playing.

Because the CD-ROM drive can handle the playing without constant program intervention, you can start the audio and then go back to using your computer normally. The audio will continue to play while you edit and compile programs or use your favorite word processor. This also allows an application to direct the CD-ROM drive to play a portion of an audio disc and then continue with other application activities. The application can check the status of the drive at any time to learn the current location of the laser pickup. This allows the application to synchronize its actions with the audio.

This chapter will discuss the software interfaces you can use to control audio playing, stopping, status inquiry, and so on. I will briefly introduce the Microsoft CD-ROM extensions. I will present CD-ROM device driver interfaces and explain how you can use various device driver services to control the playing of audio. I will illustrate how to use several of these services by including some short C language examples. I used Microsoft C v6.0 to write the C examples, but they should easily port to other C compilers. I will also demonstrate the use of the techniques discussed in the article with a simple but complete application that you can use to control audio CDs. The information contained in the article pertains to DOS, but the basic ideas apply to other operating systems (for example, OS/2) where similar software interfaces exist.

CD-ROM Software Interfaces

Software may use two major methods to access a CD-ROM device: the Microsoft CD-ROM extensions and direct calls to the CD-ROM device driver. The Microsoft CD-ROM extensions are a set of services that allow you to access a CD drive as if it were a read-only floppy disk.

The extensions are packaged in a program file called `MSCDEX.EXE`. `MSCDEX` is available through OEMs who provide CD-ROM drives; Microsoft began including `MSCDEX` in MS-DOS v6.0. You install the extensions by typing `MSCDEX` on the DOS command line (usually in your `AUTOEXEC.BAT` file). After installing `MSCDEX`, you can access the CD-ROM drive just like any other disk drive in your system. The biggest differences from other disk drives are that the CD drive is read-only and that it has a very large capacity. `MSCDEX` does not currently provide services to play audio discs. Its purpose is to allow CD-ROM devices to be used as disk drives. However, `MSCDEX` does provide several services that can be helpful when playing audio discs.

CD-ROM Device Drivers

CD-ROM drives require a device driver to interact with the operating system. The manufacturer typically provides device drivers, since CD drive hardware varies between different manufacturers. The device driver provides the "bridge" between the manufacturer-specific hardware and the operating system. Once you have developed an application that works with a particular CD-ROM device driver, there is a good chance that it will function with CD-ROM devices provided by other manufacturers.

CD-ROM device drivers provide various services for controlling audio functions as well as normal data functions. By calling these device driver services you can control the playing of audio CD discs. Audio services are available for starting audio play, stopping audio playing, resuming audio, also getting the status of the CD drive, disc, and so on.

Device Driver Basics

Device drivers for disk drives are usually block device drivers. However, CD-ROM device drivers are character device drivers. All device drivers contain a device header at the beginning of their code. The device header for a CD-ROM device looks very much like that of a standard character device driver (see Figure 25.1). The `dev_next` field contains a `far` pointer

```
typedef struct _Dev_Hdr {
        struct _Dev_Hdr _far *dev_next; // ptr-next driver
        unsigned int dev_att;   // driver attributes
        void (_near *dev_strat)(); // strategy entry
        void (_near *dev_int)();   // interrupt entry
        char dev_name[8];       // device name
        int dev_resv;           // reserved
        char dev_letr;          // 1st drive letter
        char dev_units;         // number units
        } Dev_Hdr;
```

Figure 25.1 ***Device header.***

to the next device driver in the chain of DOS devices. The `dev_att` field contains attribute bits that define various characteristics of the driver.

The `dev_strat` and `dev_int` fields contain the offsets within the device driver to the strategy and interrupt entry points. These are the entry points that other programs call to request the various services the driver provides. The `dev_letr` field tells you which drive letter DOS has assigned to the first (or only) subunit controlled by the device driver. This is a binary number where `1` corresponds to A:, `2` corresponds to B: and so on. The `dev_units` field contains a count of logical disk drives this device driver controls. These last two fields would be used if you have more than one CD-ROM device driver installed in your system and you need to find which device header corresponds to a particular logical disk drive.

When playing audio information, two fields in the device header of great interest are the `dev_strat` and `dev_int` fields. These contain the entry points you must call to request services from the device driver. These entry points are offsets into the device driver segment. By combining these offset values with the segment of the device driver, you arrive at the far addresses for the strategy and interrupt routine entry points. To request a service from the device driver you must do the following:

- build an appropriate request packet,
- point `ES:BX` to the constructed request packet,
- call the device driver strategy routine,
- call the device driver interrupt routine without disturbing the request packet, and
- check for errors.

A *request packet* consists of a request header followed by zero or more bytes of request-specific information. The exact format of the request packet depends on the service you want to request. The formats of the various request packets needed to control audio CD discs are presented later.

Locating the Header

Before you can call device driver services you need to find the device driver header. I will present two methods for doing this. The method you choose to use depends on the configuration of your particular system.

The first method involves using several services provided by the `MSCDEX` CD-ROM extensions mentioned above. You call `MSCDEX` services via `INT 2Fh`, the DOS multiplex interrupt. Microsoft assigned `MSCDEX` to function code `15h`, so you load this function code into the `AH` register and a subfunction code into the `AL` register and execute an `INT 2Fh`. The two `MSCDEX` services of immediate interest are subfunctions `0` and `1`. Subfunction `0` returns the number of installed CD-ROM drives in the `BX` register. If the `BX` register is the same after the call, you can assume that `MSCDEX` is not present. Subfunction `1` fills a buffer with a list of device driver addresses.

Listing 25.1 shows an example of using these two `MSCDEX` services to find the device header of the first CD-ROM device driver. The returned list contains one entry for each CD drive installed. Each entry consists of the device driver subunit and the far address of the device driver header. The example in Listing 25.1 demonstrates using the first installed driver, but you can easily traverse the list to get the address of any other installed driver.

The second method for finding the device driver header uses DOS services. You open the device driver using the standard DOS function for opening a file or device (`INT 21h`, function `3Dh`). If the device open is successful, DOS returns a handle. You can use this handle to perform a `READ IOCTL` function (function `44h`, subfunction `2`). CD-ROM device drivers support a `READ IOCTL` function that returns the address of the device header. The call code for this `READ IOCTL` function is `0`. To use this function you need to construct a buffer that contains the one-byte call code (in this case `0`), followed by four bytes to receive the device header address. After the `READ IOCTL` function returns, you may close the handle to the device; you no longer need it since the device header contains all the information necessary to communicate with the device. Listing 25.2 shows an example program that uses this method to get the device header for a device driver named on the command line.

Each of the two methods presented here has advantages and disadvantages. The `MSCDEX` method is very simple and results in a list of all available CD-ROM devices. You can use this method to find installed CD-ROM device drivers without having any knowledge of the device driver names. However, the `MSCDEX` method requires `MSCDEX` to be installed. If other applications require `MSCDEX`, this does not represent a penalty, but occasionally it may not be desirable to require that `MSCDEX` be installed when you only want to play audio CDs. The DOS `IOCTL` method does not require *MSCDEX*, but it does require that your program have access to the name of the device driver(s) you want to use.

If your system has more than one CD-ROM drive installed, you will need to decide which drive or drives you want to use for playing audio discs. You can examine the device header(s) to find the logical drive letter of the first logical drive and the number of logical drives supported by the driver (refer to Figure 25.1). Depending on your particular application, you might simply use the first installed drive, or you might have the user specify the desired logical drive letter or device driver name. Once you find the correct device driver header, you need to extract and save three pieces of information. You need the 32-bit strategy and interrupt entry addresses. As mentioned above, you construct these by combining the segment of the device header with the 16-bit offsets found in the strategy and interrupt entry fields in the header. The third piece of needed information is the subunit ID. A subunit ID of 0 is associated with the first device that the driver supports, subunit ID 1 is the second logical supported drive, and so on.

Requesting Services

Before requesting a device driver service you must construct a request packet. This consists of a request header followed by zero or more bytes of request-specific information. You point the *ES:BX* register pair to this request packet, call the device driver strategy entry and then call the interrupt entry. The format of the request header is shown in Figure 25.2. You must set the *rh_lth* field to the total length of the request packet.

```
typedef struct  {
      unsigned char rh_lth;   // length of request packet
      unsigned char rh_sub_unit; // target sub-unit
      unsigned char rh_cc;       // command code
      unsigned short rh_stat; // status information
      unsigned char rh_res[8];// reserved
      } Req_Head;
```

Figure 25.2 *Device driver request header.*

This is the length of the request header itself plus any additional trailing information needed for the specific request. You put the subunit ID for the drive you wish to access in the `rh_sub_unit` field.

The `rh_cc` field is filled with the command code for the specific request. The device driver sets the `rh_stat` field to show the success or failure of the request. Table 25.1 contains a partial list of valid command codes. Those marked with an asterisk are functions that were added after the initial device driver specifications. Some device drivers, particularly early versions, may not support all of the listed command codes. When dealing with audio, four of the commands listed in Table 25.1 are of interest. These are: play audio, stop audio, resume audio, and `IOCTL READ`.

Command	**Command codes in decimal**
Init	0
IOCTL input	3
Read Data (Input)	4
Non-destructive input, no wait	5
Input status	6
Input flush	7
IOCTL output	12
Device open	13
Device close	14
Read long	128*
Read long prefetch	130*
Seek	131*
Play audio	132*
Stop audio	133*
Resume audio	136*
* Denotes functions added after initial specification.	

Table 25.1 ***Device driver commands.***

Audio Play

Figure 25.3 shows the request packet needed to play audio. For this request, you must set the `rh_lth` field to `22` and the `rh_cc` field to `132`. You use the starting sector field (`p_start`) and sector length field (`p_numb`) to specify where to start the audio and how long to play. You can set the addressing mode (`p_amode`) to either High Sierra (mode 0) or Red Book (mode 1).

All CD-ROM device drivers support the High Sierra addressing mode. Many also support the Red Book mode. The High Sierra addressing mode is simply a method for addressing the sectors on the CD linearly. The Red Book addressing mode is similar to what you might see on the LCD display of a CD player. The Red Book standard expresses addressing in terms of minutes, seconds, and frames relative to the front of the disc. You can convert from the Red Book format to High Sierra according to the following formula:

```
High_Sierra = Red_Book_Frames
+ (((Red_Book_Minutes * 60)
+ Red_Book_Seconds) * 75)
```

As you can see from the formula, the CD-ROM plays 75 frames of audio every second. You can take the "least common denominator" approach and restrict your play requests to the High Sierra addressing mode. All drivers should support this mode without any problems. Listing 25.3 shows an example of a play command sequence in small model MSC v6.0.

```
typedef struct {
        Req_Head p_rh;          // request header
        unsigned char p_amode;  // addressing mode
        unsigned long p_start;  // starting sector
        unsigned long p_numb;   // number of sectors
        }  Play_packet;
```

Figure 25.3 `PLAY AUDIO` ***request packet.***

Audio Stop

You can stop audio play at any time with the stop audio function. The request packet requires no additional information beyond the request header. This service stops audio play, but does not move the laser pickup. Listing 25.4 shows an example of calling the stop function.

Resume Audio

Some drivers support the resume audio service. This service allows you to restart audio play at the spot where it was interrupted by a previous stop audio function. As with the stop function, this function requires only the request header. The calling sequence is exactly as shown in Listing 25.4, except that you use the resume audio call code (decimal 136).

Not all device drivers support this command. When working with drivers without the resume audio support, you can simulate the same action by saving the current location of the laser pickup before using the stop command. You can do this with the Q-Channel `IOCTL` inquiry (discussed later). You can then use the play audio command to resume audio play, using the saved location as the starting location.

IOCTL READ

The `IOCTL READ` function handles several subfunctions. The format of the request packet for the `IOCTL` read command is shown in Figure 25.4. The `i_md`, `i_sect`, and `i_vol` fields are unused and should be set to zero. The `i_cb` field must point to a control block. The format of the control block

```
typedef struct {
        Req_Head i_rh;               // request header
        unsigned char i_md;          // media descriptor-unused
        unsigned char _far *i_cb;    // pointer to control block
        unsigned int i_cblth;        // control block length
        unsigned int i_sect;         // start sector-unused
        unsigned char _far *i_vol;   // volid pointer-unused
        } Ioctl_rp;
```

Figure 25.4 `IOCTL READ` ***request packet.***

depends on which subfunction you are requesting. You must set the `i_cblth` field to the length of the control block, in bytes. The first byte in the control block contains the subfunction code. Table 25.2 contains a list of commonly supported subfunction codes along with the length of the control block needed for each. Figure 25.5 contains the control block format for each subfunction code discussed here.

To play from an audio disc you must decide which sectors you want to play. An audio disc is arranged as a series of tracks with each track typically containing one musical selection. The disc contains a table of contents that contains the starting sector of each audio track. You can access this information by using several subfunctions of the `IOCTL READ` function.

The `Audio Disc Info` subfunction of the `IOCTL READ` service returns the low track, high track and ending sector on the disc (see Figure 25.5). The device driver returns the ending sector value in the Red Book addressing format. The driver returns the low and high track numbers as binary numbers that range from 1 to 99.

Subfunction	Code	Lth of control block
Return device driver header address	0	5
Head location	1	6
Audio channel information	4	9
Read drive bytes	5	130
Device status	6	5
Return sector size	7	4
Return volume size	8	5
Media changed inquiry	9	2
Audio disc information	10	7
Audio track information	11	7
Audio Q-channel information	12	11
Audio Sub-channel information	13	13
UPC code	14	11
Audio status information	15	11

Table 25.2 `IOCTL READ` *subfunctions.*

You can use the `Audio Track Info` subfunction to find the starting sector number of each audio track. Place the desired track number in the `track_no` field of the control block before making the service request. The device driver sets the `st_sect` field to the starting sector in the Red

```
/* Audio Disc Info        */
typedef struct {
        unsigned char sf_di;            /* call code=10   */
        unsigned char low_track;        /* low track      */
        unsigned char hi_track;         /* high track     */
        unsigned long end_sect;         /* ending sector  */
        } Ainfo_cb;

/* Audio Track Info */
typedef struct {
        unsigned char sf_ti;            /* call code=11        */
        unsigned char track_no;         /* track number        */
        unsigned long st_sect;          /* starting sector     */
        unsigned char control_info;     /* control information*/
        } Tinfo_cb;

/* Device Status */
typedef struct {
        unsigned char sf_ds;            /* call code=6         */
        unsigned long stat;             /* status bits         */
        } DevStat_cb;

/* Media Changed */
typedef struct {
        unsigned char sf_mc;            /* call code=9         */
        char media_chnged;              /* reply code          */
        } Media_chg_cb;

/* Q-Channel Inquiry */
typedef struct {
        unsigned char sf_qc;            /* call code=12        */
        unsigned char control;          /* control info        */
        char track;                     /* current track #     */
        char point;                     /* point or index      */
                /* running time within the track        */
        char min;                       /* minute              */
        char sec;                       /* second              */
        char frame;                     /* frame               */
        char zero;                      /* unused              */
                /* running time within disc             */
        char dmin;                      /* minute              */
        char dsec;                      /* second              */
        char dframe;                    /* frame               */
        } Q_Chan_cb;
```

Figure 25.5 IOCTL READ *request packet.*

Book addressing format. The driver sets the `control_info` field with bits describing the type of information contained in the track. Listing 25.5 shows an example of using the `Audio Disc Info` and `Audio Track Info` subfunctions to get the starting sector of the first track on the disc.

The `Device Status` subfunction returns a 32-bit value that contains information about the status of the CD-ROM drive. Table 25.3 shows how to interpret the bits returned in the `stat` field (see Figure 25.5) of the control block. You can use the `Device Status` subfunction to verify that the device driver supports the playing of audio information, determine if the driver supports Red Book addressing mode, and check to see if the drive door is open.

Bits returned	`stat` **field**
Bits 31-10	Unused
Bit 9	0 = Supports high sierra addressing 1 = Supports red book and high sierra addressing
Bit 8	0 = No audio channel manipulation 1 = Supports audio channel manipulation
Bit 7	0 = No prefetch
Bit 6	1 = Prefetch supported
Bit 5	Unused
Bit 4	0 = No interleave 1 = Interleave
Bit 3	0 = Read only 1 = Read and write
Bit 2	0 = Formatted reading only 1 = Formatted and raw reading
Bit 1	0 = Door is locked 1 = Door is unlocked
Bit 0	0 = Door is closed 1 = Door is open

Table 25.3 ***Device status bits.***

CD-ROM device drivers supply a special *Media Changed IOCTL READ* subfunction. Character device drivers normally do not deal with changeable media, but CD-ROM drives are an exception. When this subfunction is called, the device driver sets the *media_chnged* byte to show if the media in the drive has changed since last checked. A value of *1* indicates that the media has not been changed, *0* means that the device driver cannot determine if the media has changed, and *-1* indicates that the media has been changed.

The last *IOCTL READ* subfunction of general interest when playing audio is the Q-Channel inquiry. This subfunction returns the current position of the laser pickup. You can use this function to provide a display similar to what you might see on a stereo CD player. The current track is returned in BCD — not binary. The minute, second, and frame values are returned in binary.

CDCNTL

Listing 25.6 contains *CDCNTL*, a Terminate-and-Stay-Resident (TSR) program written in MASM v5.1 that you can use to control CD audio discs. *CDCNTL* does not require *MSCDEX*. It uses the DOS open method to find the device driver header. I hard-wired the name of the device driver into the program at the label *DEVNAME*. You can change this to match your system or enhance the program to allow the user to specify the name on the command line.

After you install *CDCNTL*, you can activate it by depressing both shift keys and the left *CTRL* key. A pop-up window will appear if the current video mode is 2, 3, or 7. The pop-up window shows a running display of the current track, minute, and second of any disc playing. *CDCNTL* provides five functions. The *PLAY* function starts playing the disc at the first audio track. The *STOP* function stops audio play. *RESUME* starts the play at the spot where it was last stopped by the *STOP* function. The *NEXT* and *PREV* functions start playing the next and previous tracks respectively. All functions that start audio play specify a particular starting track and request playing to continue through the last audio track on the disc. The *ESC* key closes the pop-up. *CDCNTL* demonstrates most of the techniques and function calls discussed in this article.

Summary

Most programmers may consider the use of a CD-ROM drive to play audio discs to be out of the realm of mainstream PC applications. However, if you have access to a CD-ROM drive you can have a lot of fun with the ideas discussed in this article. You can also use these same techniques to add audio accompaniment to your serious applications.

For more information and further details about CD-ROM device drivers and the `MSCDEX` extensions, consult the information on the *Microsoft Programmer's Library CD*. This CD contains a section that covers these subjects in detail. Many of the low-level details in this article were originally gleaned from this source. Details regarding device drivers written by CD-ROM drive manufacturers are usually available from the specific manufacturer.

Listing 25.1 *Finding a device header using* MSCDEX *services.*

```
#include <dos.h>
#include <stdio.h>
#include <malloc.h>

#pragma pack(1)

typedef struct {    // format of device list entries
           unsigned char unit;
           unsigned short dev_header_offset;
           unsigned short dev_header_segment;
          }  dev_list;

main()
{
union REGS r;
struct SREGS sr;
dev_list *dl;

    // ask MSCDEX for # of CD drives
r.x.ax = 0x1500;
r.x.bx = 0;
int86(0x2f, &r, &r);
if (!r.x.bx)
  {
  printf("MSCDEX not installed or no CD drives\n");
  exit(1);
  }

    // allocate memory for device list
if (dl = (dev_list *) malloc(r.x.bx * sizeof (dev_list)))
  {    // fill allocated memory with device list
  segread(&sr);
  sr.es = sr.ds;
  r.x.bx = (unsigned int) dl; // list addr to es:bx
  r.x.ax = 0x1501;
  int86x(0x2f, &r, &r, &sr);  // ask MSCDEX for driver lst
  if (!r.x.cflag)
    printf("Address of 1st CD-ROM header:%04X:%04X\n",
         dl->dev_header_segment, dl->dev_header_offset);
  else
    printf("Error from MSCDEX!\n");
  }
else
  printf("Unable to allocate memory!\n");
}

// End of File
```

Listing 25.2 *Finding a device header using* `IOCTL` *services.*

```
#include <stdio.h>
#include <dos.h>
#include <fcntl.h>

#pragma pack(1)

void main(int argc, char **argv)

{
int handle;
union REGS regs;
struct {        // control block for READ IOCTL
       unsigned char cmd;
       unsigned int adr_off;
       unsigned int adr_seg;
       } addrbuf;

if (argc >= 2)
   {         // open device specified
   if (!_dos_open(argv[1], O_RDONLY, &handle))
      {
      addrbuf.cmd = 0;             // ioctl command 0
      regs.x.ax = 0x4402;          // ioctl read
      regs.x.cx = sizeof addrbuf;  // length to read
      regs.x.dx = (unsigned) &addrbuf;  // buffer to fill
      regs.x.bx = handle;          // open device handle
      intdos(&regs, &regs);        // do IOCTL read
      if (!regs.x.cflag)
         printf("Device header addr for %s is %04X:%04X\n",
                argv[1], addrbuf.adr_seg, addrbuf.adr_off);
      else
         printf("Error on IOCTL read\n");
      }
   else
      printf("Unable to open device %s\n", argv[1]);
   }
else
   printf("Please enter name of device driver\n");
}

// End of File
```

Listing 25.3 *Example of a play command sequence in small model MSC v6.0.*

```
#pragma pack(1)         // leave no extra spaces
 .
 .
Play_packet   plpack;   // request packet
unsigned long start;    // starting sector
unsigned long end;      // end sector
void (_far *strategy)();     // strategy entry
void (_far *intrpt)();       // interrupt entry
unsigned char unit;     // sub-unit ID
  .
  .
plpack.p_rh.rh_lth = sizeof plpack; // total lth
plpack.p_rh.rh_sub_unit = unit;
plpack.p_rh.rh_cc = 132;            // call code
plpack.p_amode = 0;                 // high sierra mode
plpack.p_start = start;
plpack.p_numb = end - start;
call_driver(&plpack, strategy, intrpt);  // request play
if (plpack.p_rh.rh_stat & 0x8000)
      { /* error on play */ }
else
      { /* play started OK */ }
  .
  .
void call_driver(void *pp, void (_far *strat)(), void (_far *intr)())
{
unsigned long call_strat = (unsigned long) strat;
unsigned long call_intr  = (unsigned long) intr;
unsigned int packet = (unsigned int) pp;

_asm {
     push  ds
     pop   es
     mov   bx,packet          // point es:bx to packet
     call  far ptr call_strat // call strategy entry
     call  far ptr call_intr  // call interrupt
     }
}

// End of File
```

Listing 25.4 *Example of calling the stop function.*

```
Req_Head rh;                    // request header
void (_far *strategy)();        // strategy entry
void (_far *intrpt)();          // interrupt entry
unsigned char unit;      // sub-unit ID

rh.rh_lth = sizeof rh;  // only header in packet
rh.rh_cc = 133;         // call code for stop
rh.rh_sub_unit = unit;
call_driver(&rh, strategy, intrpt);  // request stop
if (rh.rh_stat & 0x8000)
      { /* error on stop */ }
else
      { /* stopped OK */ }

// End of File
```

Listing 25.5 *Example of using the* Audio Disc Info *and* Audio Track Info *subfunctions.*

```
Ioctl_rp r_ioctl_pack;   // request packet
Ainfo_cb acb;            // control block for audio info
Tinfo_cb tcb;            // control block for track info
(_far *strategy)();      // strategy entry
(_far *intrpt)();        // interrupt entry
unsigned char unit;      // sub-unit ID

        // clear request packet
memset(&r_ioctl_pack, 0, sizeof r_ioctl_pack);

        // get audio info
r_ioctl_pack.i_rh.rh_lth = sizeof r_ioctl_pack;
r_ioctl_pack.i_rh.rh_sub_unit = unit;
r_ioctl_pack.i_rh.rh_cc = 3;   // IOCTL READ
r_ioctl_pack.i_cb = (unsigned char _far *) &acb;
acb.sf_di = 10;                // AUDIO DISC Info
call_driver(&r_ioctl_pack, strategy, intrpt);
if (r_ioctl_pack.i_rh.rh_stat & 0x8000)
   { /* error on audio info */ }
else
   {   // now get track info for low track
   tcb.track_no = acb.low_track;
   r_ioctl_pack.i_rh.rh_stat = 0;
   r_ioctl_pack.i_cb = (unsigned char _far *) &tcb;
   tcb.sf_ti = 11;             // AUDIO TRACK Info
   call_driver(&r_ioctl_pack, strategy, intrpt);
   if (r_ioctl_pack.i_rh.rh_stat & 0x8000)
     { /* error on track info */ }
   else
      { /*  start track in tcb.stsect
             in Red Book format         */ }
   }

// End of File
```

Listing 25.6 CDCNTL — a TSR to control CD audio play.

```
        TITLE   CDCNTL  TSR TO CONTROL CD AUDIO PLAY   DSB

PSP_ENV      EQU     44                 ;PSP OFFSET - ENVRN
BELL         EQU     07H                ;BELL CHARACTER
ESCKEY       EQU     27                 ;ESCAPE KEY

;*    EQUATES FOR DEVICE DRIVER COMMAND CODES                 *

PLAY_CMD        EQU  132                 ;PLAY AUDIO COMMAND
READ_IOCTL_CMD  EQU  3                   ;READ IOCTL COMMAND
STOP_CMD        EQU  133                 ;STOP AUDIO COMMAND
IOC_ADI_CMD     EQU  10                  ;AUDIO INFO IOCTL
IOC_TIF_CMD     EQU  11                  ;TRACK INFO IOCTL
IOC_ADS_CMD     EQU  6                   ;AUDIO STATUS IOCTL
IOC_MDC_CMD     EQU  9                   ;MEDIA CHANGED IOCTL
IOC_QIQ_CMD     EQU  12                  ;Q CHANNEL INQ IOCTL

;*    LAYOUT OF DEVICE DRIVER HEADER.  THIS INFO SITS AT 0 *
;*     WITHIN THE DEVICE DRIVER.                            *

DEV_HDR      STRUC                   ;DEVICE DRIVER HEADER
SDEV_NEXT    DD     ?                ;NEXT DEVICE DRIVER
SDEV_ATT     DW     ?                ;DEVICE ATTRIBUTES
SDEVSTRAT    DW     ?                ;OFFSET- STRATEGY ENTRY
SDEVINT      DW     ?                ;OFFSET- INTERRUPT ENTRY
SDEV_NAME    DB     8 DUP (?)        ;DEVICE NAME
SDEV_RESV    DW     ?                ;RESERVED
SDEV_LETR    DB     ?                ;1ST DRIVE LETTER
SDEV_NUNT    DB     ?                ;NUMBER OF SUB UNITS
DEV_HDR      ENDS

;*    LAYOUT OF DEVICE DRIVE REQUEST PACKETS.  THE REQ HDR *
;*     IS ALWAYS PRESENT.  THIS IS IMMEDIATELY FOLLOWED BY *
;*     FUNCTION SPECIFIC INFORMATION.                       *

RQH          STRUC                ;DEVICE DRIVER REQ HDR
RQH_LEN      DB     ?             ;LENGTH
RQH_UNIT     DB     ?             ;UNIT ID
RQH_CMD      DB     ?             ;COMMAND
RQH_STATUS   DW     ?             ;STATUS

RQH_RSVD     DB     8 DUP (0)     ;RESERVED
RQH          ENDS

RQP          STRUC                ;DATA FOLLOWING REQ HDR-PLAY
RQP_ADDRMD   DB     0             ;ADDRESS MODE-HIGH SIERRA=0
                                  ;             RED BOOK=1
RQP_START    DD     ?             ;START ADDR FOR PLAY
RQP_NUM      DD     ?             ;NUMBER TO PLAY
RQP          ENDS
```

Listing 25.6 *continued*

```
RQI          STRUC                 ;DATA FOLLOWING REQ HDR-IOCTL
RQI_MEDIA    DB    0               ;MEDIA - NOT USED
RQI_XFER     DD    ?               ;POINTER TO CONTROL BLOCK
RQI_NBYTES   DW    ?               ;LENGTH OF CONTROL BLOCK
RQI_SECTOR   DW    0               ;SECTOR - NOT USED
RQI_VOLID    DD    0               ;VOLID - NOT USED
RQI          ENDS

;*    CONTROL BLOCK LAYOUTS FOR VARIOUS IOCTL CALLS        *

IOS          STRUC                    ;CTRL BLK-IOCTL DVC STAT
IOS_CMD      DB    6                  ;COMMAND CODE
IOS_STAT     DD    ?                  ;DEVICE STATUS BITS
IOS          ENDS

IOS_DOOR_OPEN   EQU  0001H            ;DOOR IS OPEN
IOS_AUDIO_PLAY  EQU  0010H            ;AUDIO PLAY SUPPORTED
IOS_RED_BOOK    EQU  0200H            ;HIGH SIERRA & RED BOOK

IOM          STRUC                    ;CTRL BLK-IOCTL MEDIA CHGD
IOM_CMD      DB    9                  ;COMMAND CODE
IOM_MCHG     DB    ?                  ;MEDIA CHANGED- 1 = NO
                                      ;                0 = ??
                                      ;               -1 = CHNGD
IOM          ENDS

IOD          STRUC                    ;CTRL BLK-IOCTL AUDIO INFO
IOD_CMD      DB    10                 ;COMMAND CODE
IOD_LOTNO    DB    ?                  ;LOW TRACK NUMBER
IOD_HITNO    DB    ?                  ;HIGH TRACK NUMBER
IOD_LEADOUT  DD    ?                  ;ADDR LEADOUT (END AUDIO)
IOD          ENDS

IOT          STRUC                    ;CTRL BLK-IOCTL TRACK
INFO
IOT_CMD      DB    11                 ;COMMAND CODE
IOT_TNO      DB    ?                  ;TRACK NUMBER
IOT_START    DD    ?                  ;TRACK START
IOT_CTRL     DB    ?                  ;CONTROL INFO
IOT          ENDS
```

Listing 25.6 *continued*

```
IOQ          STRUC                  ;CTRL BLK-IOCTL Q CHANNEL
IOQ_CMD      DB    12               ;COMMAND CODE
IOQ_CTRL     DB    ?                ;CONTROL INFO
IOQ_TNO      DB    ?                ;TRACK NUMBER
IOQ_INDX     DB    ?                ;INDEX OR POINT
IOQ_MIN      DB    ?                ;MINUTES
IOQ_SEC      DB    ?                ;SECONDS
IOQ_FRAME    DB    ?                ;FRAMES
IOQ_ZERO     DB    ?                ;ZERO
IOQ_PMIN     DB    ?                ;P MINUTES
IOQ_PSEC     DB    ?                ;P SECONDS
IOQ_PFRAME   DB    ?                ;P FRAMES
IOQ          ENDS

;*    LAYOUT OF TRACK INFO ARRAY ENTRIES IN 'TOCINFO'      *

TOCI         STRUC                  ;TOC TRACK INFO ENTRIES
TOC_TNO      DB    ?                ;TRACK NUMBER
TOC_START    DD    ?                ;START ADDR
TOCI         ENDS

DATA    SEGMENT PARA PUBLIC

STACKINT9 DW    256 DUP (?)     ;STACK USED FOR POPUP
SVSPINT9 DW     ?               ;SAVE FOR STACK POINTER
SVSSINT9 DW     ?               ;SAVE FOR STACK SEGMENT
REGENADR DW     ?               ;VIDEO MEMORY SEG FOR I/O

                                  ;CURRENT CD INFO
TNO     DB      0                 ;TRACK NUMBER
TIMEM   DB      0                 ;MINUTE
TIMES   DB      0                 ;SECONDS
CLOSED  DB      0                 ;DOOR CURRENTLY CLOSED
                                  ; 0 = NO, 1 = YES

TOCINFO TOCI    99 DUP (<>)     ;SPACE FOR 99 AUDIO TRACKS
LOWTNO  DB      ?               ;LOW TRACK ON DISC
HIGHTNO DB      ?               ;HIGH TRACK ON DISC
ENDADDR DD      ?               ;ENDING ADDR

DEV_STRAT DD    ?               ;STRATEGY ENTRY POINT
DEV_INT   DD    ?               ;INTERRUPT ENTRY POINT
DEV_UNIT  DB    0               ;SUB UNIT ID
```

Listing 25.6 *continued*

```
                                ;DEVICE DRIVER REQ PACKET
REQHDR  RQH     <>              ;REQUEST HEADER AREA
REQFLW  RQI     <>              ;DATA FOLLOWING THE REQ HDR
                                ; (FUNCTION DEPENDENT)

IOCTLB  IOQ     <>              ;AREA FOR IOCTL CONTROL BLK

CVTWK   DW      ?               ;WORK AREA

POP_COLS      EQU    40
POP_ROWS      EQU    12
POP_START_ROW EQU    7
POP_START_COL EQU    21
POP_BG        EQU    30H
            ;SAVE AREA FOR VIDEO MEMORY
POP_SAVE    DW   POP_COLS*POP_ROWS DUP (?)

REGEN       LABEL DWORD
REGENO      DW   4EH
REGENS      DW   40H            ;ADDR OF VIDEO REGEN AREA

VMODE       LABEL DWORD
VMODEO      DW   49H
VMODES      DW   40H            ;ADDR OF VIDEO MODE

POP_IMG DB      ''
        DB      '          CD-ROM AUDIO CONTROL         '
        DB      '        CURRENT TRACK '
P_TNO   DB      ' 0'
        DB      '  '
P_TIME  DB      5 DUP (' ')
        DB      '        '
        DB      '         SELECT A FUNCTION:            '
        DB      '            P - Play                   '
        DB      '            S - Stop                   '
        DB      '            R - Resume                 '
        DB      '            N - Next                   '
        DB      '            L - Prev                   '
        DB      '           ESC- CLOSE WINDOW           '
        DB      '                                       '
        DB      ''

DATA    ENDS
```

Listing 25.6 *continued*

```
CODE    SEGMENT PARA PUBLIC
        ASSUME  CS:CODE,DS:DATA

;*              INT09H INTERRUPT HANDLER                   *
;*             DO POPUP ON SHIFT,SHIFT,CTRL                *

OLD_INT9 DD     ?               ;SAVED INT 09
IN_POPUP DB     0               ;NON-ZERO IF POPUP ACTIVE
SIG     DB      'CDCNTL'        ;SIGNATURE -
SIGL    EQU     $-SIG           ;RIGHT BEFORE INT9 ENTRY

TRAP_KB PROC    FAR             ;PROCESS INT09 KEYBOARD HIT
        CMP     IN_POPUP,0      ;CHK ALREADY IN POPUP
        JE      NOTINPOP
        JMP     OLD_INT9        ;CONT TO NEXT HANDLER-ACTIVE
NOTINPOP:
        MOV     IN_POPUP,1
        PUSH    DS
        PUSH    BX
        MOV     BX,DATA
        MOV     DS,BX           ;SET UP DS
        MOV     SVSPINT9,SP     ;SAVE INCOMING SS:SP
        MOV     BX,SS
        MOV     SVSSINT9,BX
        MOV     BX,DS
        MOV     SS,BX           ;SWITCH TO LOCAL STACK
        MOV     SP,OFFSET STACKINT9+SIZE STACKINT9-2
        PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    DI
        PUSH    SI
        PUSH    ES
        PUSH    BP
        PUSHF
        CALL    OLD_INT9         ;CALL OLD KEYBOARD ROUT
        MOV     AH,2
        INT     16H              ;GET SHIFT STATUS
        AND     AL,07H
        CMP     AL,07H           ;CHK FOR CTRL, SHIFT,SHIFT
        JNE     KBRTN
```

Listing 25.6 *continued*

```
        JMP     DOPOPUP
KBRTN:
        POP     BP
        POP     ES
        POP     SI
        POP     DI
        POP     DX
        POP     CX
        POP     AX
        MOV     BX,SVSSINT9
        CLI
        MOV     SS,BX
        MOV     SP,SVSPINT9       ;SWITCH BACK TO SAVED STACK
        POP     BX
        POP     DS
        MOV     IN_POPUP,0
        IRET
DOPOPUP:
        MOV     DX,0B800H
        LES     BX,VMODE
        MOV     AL,ES:[BX]        ;GET CURRENT VIDEO MODE
        CMP     AL,2
        JE      OKVIDEOM
        CMP     AL,3
        JE      OKVIDEOM
        MOV     DX,0B000H
        CMP     AL,7
        JE      OKVIDEOM
        JMP     KBRTN             ;IGNORE-NOT IN MODE 2,3,7
OKVIDEOM:
        CLD
        LES     BX,REGEN
        MOV     AX,ES:[BX]        ;GET START VIDEO REGEN AREA
        MOV     CL,4
        SHR     AX,CL             ;CALC PARAGRAPH
        ADD     AX,DX             ;ADD BASE VIDEO SEGMENT
        MOV     REGENADR,AX       ;SAVE SEGMENT FOR VIDEO I/O
        MOV     ES,AX
        MOV     AX,POP_START_ROW
        DEC     AX
        MOV     CX,160            ;BYTES/ROW OF VIDEO MEMORY
        MUL     CL
        MOV     SI,AX
        MOV     AX,POP_START_COL
        DEC     AX
        SHL     AX,1              ;2 BTYES/COL
        ADD     SI,AX             ;1ST MEMORY AREA TO SAVE
        MOV     DI,OFFSET POP_SAVE
        PUSH    DS
        PUSH    ES
        POP     DS
        POP     ES
        MOV     CX,POP_ROWS       ;NUMBER OF ROWS TO SAVE
```

Listing 25.6 *continued*

```
SAVVID:
        PUSH    CX
        PUSH    SI

        MOV     CX,POP_COLS     ;NUMBER OF COLS TO SAVE
        REP     MOVSW           ;SAVE ROW OF VIDEO
        POP     SI
        POP     CX
        ADD     SI,160          ;GO NEXT ROW
        LOOP    SAVVID
        MOV     AX,DATA
        MOV     DS,AX           ;RESTORE DS
        CALL    FILL_CHECKING   ;SHOW CHECKING STATUS
CALLPOP:                        ;  ***MAIN INPUT LOOP***
        CALL    PUT_POP
        MOV     AH,1
        INT     16H             ;CHK KEYSTROKE WAITING
        JNZ     KEYIN
        CALL    CHKCHG          ;GET CURRENT AUDIO STATUS
        MOV     AL,TNO
        MOV     SI,OFFSET P_TNO
        CALL    CVHEX           ;CONVERT TRACK NUMBER
        PUSH    DS
        POP     ES
        MOV     DI,OFFSET P_TIME
        MOV     AL,' '
        MOV     CX,SIZE P_TIME
        REP     STOSB           ;CLEAR TIME AREA
        CMP     TNO,0
        JE      DUNTIMEZ
        MOV     AL,TIMEM
        MOV     SI,OFFSET P_TIME
        CALL    CVHEX           ;CONVERT MINUTES
        MOV     AL,TIMES
        MOV     SI,OFFSET P_TIME+3
        CALL    CVHEX
        CMP     P_TIME+3,' '
        JNE     DUNTIME
        MOV     P_TIME+3,'0'    ;ZERO FILL SECONDS
DUNTIME:
        MOV     P_TIME+2,':'
DUNTIMEZ:
        JMP     CALLPOP         ;CONTINUE TO UPDATE POPUP INFO
KEYIN:                          ;PROCESS KEYSTROKE
        MOV     AH,0
        INT     16H             ;GET QUEUED KEYSTROKE
        CMP     AL,ESCKEY
        JNE     FCT_IN
GORESTV:
        JMP     REST_VIDEO
```

Listing 25.6 *continued*

```
FCT_IN:
        OR      AL,20H          ;MAKE LOWER CASE
        CMP     AL,'p'          ;CHK FOR SUPPORTED FCTS
        JE      CHK_FCTIN
        CMP     AL,'s'
        JE      CHK_FCTIN
        CMP     AL,'r'
        JE      CHK_FCTIN
        CMP     AL,'n'
        JE      CHK_FCTIN
        CMP     AL,'l'
        JE      CHK_FCTIN
BELL_AGAIN:
        MOV     AL,BELL
        MOV     AH,0EH
        INT     10H
        JMP     CALLPOP
CHK_FCTIN:
        CMP     CLOSED,0        ;CHK DOOR OPEN
        JNE     DOOR_CLOSED
        JMP     BELL_AGAIN
DOOR_CLOSED:
        CMP     AL,'p'          ;CHK PLAY
        JNE     NOT_PLAY
        MOV     AL,LOWTNO       ;START WITH LOW TRACK#
CALL_PLAY:
        CALL    PLAY            ;PLAY AUDIO
        OR      AX,AX
        JNZ     BELL_AGAIN
        JMP     CALLPOP
NOT_PLAY:
        CMP     AL,'r'          ;CHK RESUME
        JNE     NOT_RESUME
        CALL    RESUME          ;RESUME PLAY
        OR      AX,AX
        JNZ     BELL_AGAIN
        JMP     CALLPOP
NOT_RESUME:
        CMP     AL,'s'          ;CHK STOP
        JNE     NOT_STOP
        CALL    STOP            ;STOP PLAYING
        JMP     CALLPOP
```

Listing 25.6 *continued*

```
NOT_STOP:
        CMP     AL,'n'          ;CHK NEXT TRACK
        JNE     NOT_NEXT
        CMP     LOWTNO,0
        JE      BELL_AGAIN
        MOV     AL,TNO
        OR      AL,AL
        JZ      BELL_AGAIN
        INC     AL
        CMP     AL,HIGHTNO      ;CHK THERE IS A NEXT TRACK
        JG      BELL_AGAIN
        JMP     CALL_PLAY
NOT_NEXT:                       ;PREV TRACK REQUEST
        CMP     LOWTNO,0
        JE      BELL_AGAIN
        MOV     AL,TNO
        OR      AL,AL

        JZ      BELL_AGAIN
        DEC     AL
        CMP     AL,LOWTNO       ;CHK THERE IS A PREV TRACK
        JL      BELL_AGAIN
        JMP     CALL_PLAY
REST_VIDEO:                     ;RESTORE SAVE VIDEO IMAGE
        MOV     AX,REGENADR
        MOV     ES,AX
        MOV     AX,POP_START_ROW
        DEC     AX
        MOV     CX,160          ;BYTES/ROW OF VIDEO MEMORY
        MUL     CL
        MOV     DI,AX
        MOV     AX,POP_START_COL
        DEC     AX
        SHL     AX,1            ;2 BTYES/COL
        ADD     DI,AX           ;1ST MEMORY AREA TO RESTORE
        MOV     SI,OFFSET POP_SAVE
        MOV     CX,POP_ROWS     ;NUMBER OF ROWS SAVED
RESTVID:
        PUSH    CX
        PUSH    DI
        MOV     CX,POP_COLS     ;NUMBER OF COLS TO REST
        REP     MOVSW           ;REST ROW OF VIDEO
        POP     DI
        POP     CX
        ADD     DI,160          ;GO NEXT ROW
        LOOP    RESTVID
        JMP     KBRTN
TRAP_KB ENDP
```

Listing 25.6 *continued*

```
;*      ROUTINE TO CONVERT NUMBER IN AL TO DS:SI   *
;*        INPUT IS IN BINARY                       *
;*        CONVERTED TO ASCII CHARACTERS            *

CVHEX   PROC
        PUSH    AX
        OR      AL,AL           ;CHK ZERO
        JNZ     CVHEXA
        MOV     AX,'0 '
        JMP     CVHEXZ
CVHEXA:
        MOV     AH,0
        PUSH    CX
        MOV     CL,10
        DIV     CL
        ADD     AX,'00'         ;DISPLAY IN ASCII
        POP     CX
        CMP     AL,'0'

        JNE     CVHEXZ
        MOV     AL,' '          ;BLANK 1ST DIGIT IF ZERO
CVHEXZ:
        MOV     [SI],AX         ;MOVE TO OUTPUT POSITION
        POP     AX
        RET
CVHEX   ENDP

;* ROUTINE TO FILL POPUP TO SHOW CHECKING DISC STATUS      *

FILL_CHECKING   PROC
        MOV     WORD PTR P_TNO,'0 '      ;TRACK ZERO
        MOV     WORD PTR P_TIME,'AW'     ;FILL 'WAIT'
        MOV     WORD PTR P_TIME+2,'TI'
        MOV     P_TIME+4,' '
        RET
FILL_CHECKING   ENDP
```

Listing 25.6 *continued*

```
;* ROUTINE TO PAINT THE POPUP TO THE SCREEN                *

PUT_POP PROC
        MOV     AX,REGENADR
        MOV     ES,AX
        MOV     AX,POP_START_ROW
        DEC     AX
        MOV     CX,160          ;BYTES/ROW OF VIDEO MEMORY
        MUL     CL
        MOV     DI,AX
        MOV     AX,POP_START_COL
        DEC     AX
        SHL     AX,1            ;2 BTYES/COL
        ADD     DI,AX           ;1ST MEMORY AREA TO PAINT
        MOV     SI,OFFSET POP_IMG
        MOV     CX,POP_ROWS     ;NUMBER OF ROWS TO PAINT
PUTVIDA:
        PUSH    CX
        PUSH    DI
        MOV     CX,POP_COLS     ;NUMBER OF COLS TO PAINT
PUTVIDB:
        LODSB
        MOV     AH,POP_BG       ;ATTRIBUTE
        STOSW
        LOOP    PUTVIDB
        POP     DI
        POP     CX
        ADD     DI,160          ;GO NEXT ROW

        LOOP    PUTVIDA
        RET
PUT_POP ENDP

        PAGE
```

Listing 25.6 *continued*

```
;* ROUTINE TO DO AN IOCTL CALL                             *
;* MUST FILL IOCTL CONTROL BLOCK AND LTH TO RQI_NBTYES     *
;* AX RETURN NON-ZERO WITH STATUS IF ERROR OCCURS          *

IOCTL   PROC
        PUSH   ES
        PUSH   BX
        MOV    REQHDR.RQH_LEN,TYPE RQH + TYPE RQI ;LTH REQ
        MOV    AL,DEV_UNIT
        MOV    REQHDR.RQH_UNIT,AL      ;SUB UNIT ID
        MOV    REQHDR.RQH_CMD,READ_IOCTL_CMD ;READ IOCTL
        MOV    REQHDR.RQH_STATUS,0
        MOV    REQFLW.RQI_MEDIA,0      ;MEDIA - NOT USED
        MOV    REQFLW.RQI_SECTOR,0     ;SECTOR - NOT USED
        MOV    WORD PTR REQFLW.RQI_VOLID,0 ;VOLID - UNUSED
        MOV    WORD PTR REQFLW.RQI_VOLID+2,0
        PUSH   DS
        POP    WORD PTR REQFLW.RQI_XFER+2 ;ADDR OF CTL BLK
        MOV    WORD PTR REQFLW.RQI_XFER,OFFSET IOCTLB
        PUSH   DS
        POP    ES
        MOV    BX,OFFSET REQHDR  ;POINT ES:BX TO REQ HDR
        CALL   CALL_DVC          ;CALL DEVICE DRIVER
        POP    BX                ; ERROR CODE IN AX
        POP    ES
        RET
IOCTL   ENDP

;*  CALL DEVICE DRIVER.  ES:BX POINTS TO REQUEST PACKET.   *
;*                AX RETURNS 0 IF OK, ELSE ERROR CODE.      *

CALL_DVC PROC
        CALL   DEV_STRAT       ;CALL STATEGY ENTRY
        CALL   DEV_INT         ;CALL INTERRUPT ENTRY
        MOV    AX,0            ;SHOW NO ERROR
        TEST   REQHDR.RQH_STATUS,8000H  ;CHK ERROR
        JZ     CALL_DVCZ
        MOV    AX,REQHDR.RQH_STATUS
CALL_DVCZ:
        RET
CALL_DVC ENDP
```

Listing 25.6 *continued*

```
;*      ROUTINE TO FILL THE TOC INFO TO TOCINFO            *
;*              RETURN AX NON-ZERO IF ERROR                *

TOC     PROC
        PUSH    CX
        PUSH    SI
        MOV     LOWTNO,0        ;INDICATES NO DISC
        MOV     IOCTLB.IOD_CMD,IOC_ADI_CMD ;FILL CTRL BLK-
        MOV     REQFLW.RQI_NBYTES,TYPE IOD ; AUDIO DISK INFO
        CALL    IOCTL                      ;GET AUDIO INFO
        OR      AX,AX
        JNZ     TOCZ
        MOV     AL,IOCTLB.IOD_LOTNO     ;GET LOW, HIGH TNOS
        MOV     LOWTNO,AL
        MOV     AL,IOCTLB.IOD_HITNO
        MOV     HIGHTNO,AL
        MOV     AX,WORD PTR IOCTLB.IOD_LEADOUT  ;GET END
        MOV     WORD PTR ENDADDR,AX             ; ADDRESS
        MOV     AX,WORD PTR IOCTLB.IOD_LEADOUT+2
        MOV     WORD PTR ENDADDR+2,AX
        MOV     SI,OFFSET TOCINFO       ;FILL TRACK INFO
        MOV     CL,LOWTNO
        MOV     IOCTLB.IOT_CMD,IOC_TIF_CMD
        MOV     REQFLW.RQI_NBYTES,TYPE IOT
TOCA:
        CMP     CL,HIGHTNO              ;CHK FINISHED
        JNG     TOCB
        MOV     AX,0
        JMP     TOCZ
TOCB:
        MOV     IOCTLB.IOT_TNO,CL
        CALL    IOCTL                   ;DO IOCTL-TRACK INFO
        OR      AX,AX
        JZ      TOCC
        MOV     LOWTNO,0                ;MARK NO DISC IF ERR
        JMP     TOCZ
TOCC:
        MOV     [SI].TOC_TNO,CL         ;FILL TRACK NUMBER
        MOV     AX,WORD PTR IOCTLB.IOT_START
        MOV     WORD PTR [SI].TOC_START,AX      ;START ADDR
        MOV     AX,WORD PTR IOCTLB.IOT_START+2
        MOV     WORD PTR [SI].TOC_START+2,AX
        ADD     SI,TYPE TOCI             ;GO NEXT TRACK
        INC     CL
        JMP     TOCA
TOCZ:
        POP     SI
        POP     CX
        RET
TOC     ENDP
```

Listing 25.6 *continued*

```
;*      ROUTINE TO GET CURRENT STATUS OF DISC           *
;*      REREAD TOC IF NEEDED.  AX RETURN NON-ZERO IF ERR *
CHKCHG  PROC
        PUSH    BX
        PUSH    DX
        MOV     TNO,0           ;INDICATES NO DISC OR ERROR
        MOV     TIMEM,0
        MOV     TIMES,0
        MOV     CLOSED,0
        MOV     IOCTLB.IOS_CMD,IOC_ADS_CMD ;SET FOR DEVICE
        MOV     REQFLW.RQI_NBYTES,TYPE IOS ; STATUS IOCTL
        CALL    IOCTL           ;DO IOCTL
        OR      AX,AX
        JZ      CHKCHGA
        JMP     CHKCHGZ
CHKCHGA:
        TEST    WORD PTR IOCTLB.IOS_STAT,IOS_DOOR_OPEN
        JZ      CHKCHGB
        JMP     CHKCHGZ         ;RETURN - DOOR OPEN
CHKCHGB:
        MOV     CLOSED,1        ;DOOR IS CLOSED
        CALL    Q_INQ           ;DO A Q CHANNEL INQ
        OR      AX,AX
        JZ      CHKCHGC
        JMP     CHKCHGZ         ;LEAVE IF ERROR
CHKCHGC:
        MOV     TNO,DH          ;FILL TRACK,
        MOV     TIMEM,DL        ; MINUTES AND SECONDS
        MOV     TIMES,BH
        CMP     TNO,0           ;CHK PLAYING
        JE      CHKCHGF
        JMP     CHKCHGZ         ; YES - RETURN
CHKCHGF:
        MOV     IOCTLB.IOM_CMD,IOC_MDC_CMD ;SET FOR
        MOV     REQFLW.RQI_NBYTES,TYPE IOM ; IOCTL MEDIA CHNG
        CALL    IOCTL
        OR      AX,AX
        JZ      CHKCHGG
        JMP     CHKCHGZ
CHKCHGG:
        CMP     IOCTLB.IOM_MCHG,1       ;CHK MEDIA NOT CHNGD
        JNE     CHKCHGH
        JMP     CHKCHGZ                 ; NO - RETURN
CHKCHGH:
        CALL    FILL_CHECKING           ; YES - GET READY
        CALL    PUT_POP                 ;  TO RE-READ TOC
        CALL    TOC                     ;RE READ TOC
        MOV     TNO,0
```

Listing 25.6 *continued*

```
        MOV     TIMEM,0
        MOV     TIMES,0
        CALL    Q_INQ                   ;DO Q CHAN INQ
        OR      AX,AX                   ; AFTER TOC
        JZ      CHKCHGI
        JMP     CHKCHGZ
CHKCHGI:
        MOV     TNO,DH
        MOV     TIMEM,DL
        MOV     TIMES,BH
        MOV     AX,0
CHKCHGZ:
        POP     DX
        POP     BX
        RET
CHKCHG  ENDP

;*      DO Q CHANNEL INQ.                                 *
;*   AX RETURNS ZERO IF SUCCESSFUL                        *
;*   IF SUCCESSFUL - DH = TNO, DL = MINUTES, BH = SECONDS, *
;*                          BL = FRAME                    *

Q_INQ   PROC
        MOV     IOCTLB.IOQ_CMD,IOC_QIQ_CMD ;SET FOR
        MOV     REQFLW.RQI_NBYTES,TYPE IOQ ; Q CHANNEL INQ
        CALL    IOCTL
        OR      AX,AX
        JZ      Q_INQA
        RET                     ;RETURN ERROR
Q_INQA:
        MOV     AL,IOCTLB.IOQ_TNO
        CALL    CVTTNO          ;CONVERT TNO TO BINARY
        MOV     DH,AL           ;FILL TRACK, MIN, SEC
        MOV     DL,IOCTLB.IOQ_MIN
        MOV     BH,IOCTLB.IOQ_SEC
        MOV     BL,IOCTLB.IOQ_FRAME
        XOR     AX,AX
        RET
Q_INQ   ENDP
```

Listing 25.6 *continued*

```
;*     CONVERT BCD IN AL TO BINARY IN AX                          *

CVTTNO  PROC
        PUSH    BX
        PUSH    CX
        MOV     BL,AL

        AND     BL,0FH          ;GET LOW NIBBLE TO BL
        MOV     CL,4
        SHR     AL,CL           ;GET HIGH NIBBLE TO AL
        MOV     CL,10
        MUL     CL
        MOV     BH,0            ;MULT HIGH NIBBLE BY 10
        ADD     AX,BX
        POP     CX              ;COMBINE NIBBLES
        POP     BX
        RET
CVTTNO  ENDP

;*      HIGH-LEVEL PLAY ROUTINE                                    *
;*        START TRACK IN AL                                        *
;*        AX RETURN NON-ZERO IF ERROR                              *

PLAY    PROC
        PUSH    SI
        PUSH    CX
        PUSH    ES
        PUSH    BX
        CMP     AL,LOWTNO       ;CHK GOOD TRACK NUMBER
        JL      PLAYA
        CMP     AL,HIGHTNO
        JNA     PLAYB
PLAYA:
        MOV     AX,1
        JMP     PLAYZ
PLAYB:
        MOV     SI,OFFSET TOCINFO
PLAYC:
        CMP     AL,[SI].TOC_TNO ;LOOK UP TRACK NUMBER
        JE      PLAYD
        ADD     SI,TYPE TOCI
        JMP     PLAYC
PLAYD:
        MOV     AX,WORD PTR [SI].TOC_START   ;START ADDR
        MOV     BX,WORD PTR [SI].TOC_START+2 ; TO BX:AX
        CALL    CVTHSG          ;CONVERT TO HIGH SIERRA
        CALL    PLAYLOW         ;GO PLAY FROM BX:AX TO END
```

Listing 25.6 *continued*

```
PLAYZ:
        POP     BX
        POP     ES
        POP     CX
        POP     SI
        RET
PLAY    ENDP

;*      LOW-LEVEL PLAY ROUTINE                          *
;* PLAY FROM ADDR IN BX:AX (HIGH SIERRA FMT) TO LAST SECT *
;*             AX RETURNED NON-ZERO IF ERROR            *

PLAYLOW PROC
        PUSH    DX
        PUSH    SI
        PUSH    CX
        PUSH    ES
        PUSH    AX
        CALL    STOP
        POP     AX
        MOV     REQHDR.RQH_LEN,TYPE RQH+TYPE RQP ;LTH PACKET
        MOV     DL,DEV_UNIT
        MOV     REQHDR.RQH_UNIT,DL      ;SUB UNIT ID
        MOV     REQHDR.RQH_CMD,PLAY_CMD ;PLAY COMMAND
        MOV     REQHDR.RQH_STATUS,0
        MOV     REQFLW.RQP_ADDRMD,0     ;HIGH SIERRA MODE
        MOV     WORD PTR REQFLW.RQP_START+2,BX
        MOV     WORD PTR REQFLW.RQP_START,AX  ;FILL START
        PUSH    BX
        PUSH    AX
        MOV     BX,WORD PTR ENDADDR+2
        MOV     AX,WORD PTR ENDADDR
        CALL    CVTHSG          ;CONVERT END ADDR
        POP     SI              ; (BX:AX) TO HIGH SIERRA
        POP     CX              ;GET START(HIGH SIERRA)-CX:SI
        SUB     AX,SI           ;CALC NUMBER FRAMES
        SBB     BX,CX
        MOV     WORD PTR REQFLW.RQP_NUM,AX  ;FILL NUM
        MOV     WORD PTR REQFLW.RQP_NUM+2,BX
        PUSH    DS
        POP     ES
        MOV     BX,OFFSET REQHDR        ;ES:BX TO REQ HDR
        CALL    CALL_DVC                ;START AUDIO PLAY
PLAYLOWZ:
        POP     ES
        POP     CX
        POP     SI
        POP     DX
        RET
PLAYLOW ENDP
```

Listing 25.6 *continued*

```
;*              ROUTINE TO STOP AUDIO                      *
;*              AX RETURN NON-ZERO IF ERROR                *

STOP    PROC
        PUSH    ES
        PUSH    BX
        MOV     REQHDR.RQH_LEN,TYPE RQH ;SIZE OF HEADER
        MOV     AL,DEV_UNIT

        MOV     REQHDR.RQH_UNIT,AL      ;SUB UNIT ID
        MOV     REQHDR.RQH_CMD,STOP_CMD ;STOP COMMAND
        MOV     REQHDR.RQH_STATUS,0
        PUSH    DS
        POP     ES
        MOV     BX,OFFSET REQHDR        ;ES:BX TO REQ HDR
        CALL    CALL_DVC                ;CALL DEVICE DRIVER
        POP     BX
        POP     ES
        RET
STOP    ENDP

;*              ROUTINE TO RESUME AUDIO                    *
;*              AX RETURN NON-ZERO IF ERROR                *

RESUME  PROC
        PUSH    ES
        PUSH    BX
        PUSH    DX
        PUSH    SI
        CALL    Q_INQ           ;GET Q CHANNEL INFO TO DX:BX
        OR      AX,AX
        JZ      RESUMEB
        JMP     RESUMEZ
RESUMEB:
        CMP     DH,0            ;CHK TNO == 0
        JNE     RESUMEC         ; YES - CAN'T RESUME
        MOV     AX,1
        JMP     RESUMEZ
RESUMEC:
        MOV     AX,BX
        MOV     BX,DX           ;GET CURRENT POS TO BX:AX
        CALL    CVTHSG          ;CONVERT TO HIGH SIERRA THIS IS OFFSET INTO TRACK
        CMP     DH,LOWTNO       ;CHK GOOD TRACK NUMBER
        JL      RESUMED
        CMP     DH,HIGHTNO
        JNA     RESUMEE
RESUMED:
        MOV     AX,1
        JMP     RESUMEZ
RESUMEE:
        PUSH    AX
        PUSH    BX              ;SAVE HIGH SIERRA OFFSET
        MOV     SI,OFFSET TOCINFO
```

Listing 25.6 *continued*

```
RESUMEF:
        CMP     DH,[SI].TOC_TNO ;LOOK UP TRACK NUMBER
        JE      RESUMEG
        ADD     SI,TYPE TOCI
        JMP     RESUMEF

RESUMEG:
        MOV     AX,WORD PTR [SI].TOC_START ;START ADDR-BX:AX
        MOV     BX,WORD PTR [SI].TOC_START+2
        CALL    CVTHSG          ;CONVERT IT TO HIGH SIERRA
        POP     DX
        POP     SI              ;GET SAVED OFFSET TO DX:SI
        ADD     AX,SI
        ADC     BX,DX           ;ADD TO FRONT OF TRACK
        CALL    PLAYLOW         ;PLAY FROM HERE TO END
RESUMEZ:
        POP     SI
        POP     DX
        POP     BX
        POP     ES
        RET
RESUME  ENDP

;* ROUTINE TO CONVERT REDBOOK TO HIGH SIERRA ADDR IN        *
;*      BX:AX                                               *
;* REDBOOK INPUT:  BL=MINUTES, AH=SECONDS, AL=FRAMES        *
;* HIGH SIERRA = ((MINUTES * 60) + SECONDS) * 75 + FRAMES   *

CVTHSG  PROC
        PUSH    CX
        PUSH    DX
        PUSH    AX      ;SAVE LOW ORDER
        MOV     BH,0    ;MSB SHOULD BE ZERO
        MOV     AX,BX
        MOV     CL,60
        MUL     CL      ;GET SECONDS TO AX
        POP     BX      ;GET LOW ORDER
        PUSH    BX
        MOV     BL,BH   ;GET SECONDS
        MOV     BH,0
        ADD     AX,BX   ;ADD SECONDS
        MOV     CX,75
        MUL     CX      ;GET FRAMES TO DX:AX
        POP     BX
        MOV     BH,0    ;GET FRAMES
        ADD     AX,BX   ;ADD FRAMES
        ADC     DX,0
        MOV     BX,DX   ;GET RESULT TO BX:AX
        POP     DX
        POP     CX
        RET
CVTHSG  ENDP
```

Listing 25.6 *continued*

```
KEEP_MEM LABEL  BYTE    ;MEMORY BELOW DISCARDED ON TSR

        PAGE

;*   ROUTINE TO INSTALL CDCNTL                             *
;*   THIS ROUTINE GETS CONTROL WHEN CDCNTL CALLED FROM DOS *

PSP     DW      ?    ;TEMP SAVE FOR PSP VALUE

DEVNAME DB      'CDROM1',0      ;DEVICE TO OPEN

INSTALL_MSG DB 'CD-ROM AUDIO CONTROL',0DH,0AH
            DB 'DEPRESS LEFT CTRL AND BOTH SHIFTS TO '
            DB ' ACTIVATE',0DH,0AH,'$'
STATE_MSG   DB 'CDCNTL ALREADY INSTALLED - IGNORED.'
            DB 0DH,0AH,BELL,'$'
INSUFDD     DB 'DEVICE DRIVER DOES NOT SUPPORT AUDIO '
            DB 'PLAY COMMANDS - CDCNTL'
            DB ' NOT INSTALLED!',0DH,0AH,BELL,'$'
BADFIOCTL   DB 'ERROR ON IOCTL - CDCNTL NOT INSTALLED!'
            DB 0DH,0AH,BELL,'$'
ERROPEN_MSG DB 'UNABLE TO OPEN CDROM1 DEVICE - CDCNTL '
            DB 'NOT INSTALLED!'
            DB 0DH,0AH,BELL,'$'
IOCTL_PACKET DB 0       ;GET DEVICE HEADER ADDR COMMAND
IOCTL_PACKAD DD ?       ;DEVICE DRIVER ADDRESS FILLED HERE
IOCTL_PACKETL EQU  $-IOCTL_PACKET

CDCNTL  PROC    FAR
        CLD
        MOV     AX,DATA
        MOV     DS,AX
        MOV     PSP,ES          ;SAVE PSP SEGMENT
        MOV     AX,3509H
        INT     21H             ;GET CURRENT INT09H
        MOV     DI,BX
        SUB     DI,SIGL         ;BACKUP UP TO SIG AREA
        MOV     SI,OFFSET SIG
        MOV     CX,SIGL
        PUSH    DS
        PUSH    CS
        POP     DS
        REP     CMPSB           ;CHK ALREADY INSTALLED
        POP     DS
        JE      PUTSTATE
        JMP     DOINSTALL
```

Listing 25.6 *continued*

```
PUTSTATE:
        MOV     DX,OFFSET STATE_MSG
        MOV     BL,1    ;END ERRORLEVEL 1-ALREADY INSTALLED
ERROR_OUT:
        PUSH    DS
        PUSH    CS
        POP     DS
        MOV     AH,9
        INT     21H             ;ALREADY INSTALLED MSG
        POP     DS
        MOV     AH,4CH
        MOV     AL,BL
        INT     21H             ;NORM EOJ-ERROR LEVEL SET
DOINSTALL:
        PUSH    ES              ;SAVE INT9 VECTOR VALUE
        PUSH    BX
        MOV     DX,OFFSET DEVNAME
        MOV     AX,3D00H
        PUSH    DS
        PUSH    CS
        POP     DS
        INT     21H             ;TRY TO OPEN CDROM DEVICE
        POP     DS
        JNC     OKDEV_OPEN
        MOV     DX,OFFSET ERROPEN_MSG
        MOV     BL,2            ;ERRORLEVEL 2-CANT OPEN
        JMP     ERROR_OUT
OKDEV_OPEN:
        MOV     BX,AX           ;HANDLE TO DEVICE
        MOV     DX,OFFSET IOCTL_PACKET
        MOV     CX,IOCTL_PACKETL
        MOV     AX,4402H
        PUSH    DS
        PUSH    CS
        POP     DS
        INT     21H             ;DO GENERIC READ IOCTL CALL
        POP     DS
        JNC     IOCTL_OK
        MOV     DX,OFFSET BADFIOCTL
        MOV     BL,3            ;ERRORLEVEL3-ERROR DOS IOCTL
        JMP     ERROR_OUT
```

Listing 25.6 *continued*

```
IOCTL_OK:
        MOV     AH,3EH
        INT     21H             ;CLOSE DEVICE HANDLE
        LES     BX,IOCTL_PACKAD ;GET ADDR OF DEVICE HEADER
        MOV     AX,ES
        MOV     WORD PTR DEV_STRAT+2,AX ;GET ENTRIES FOR
        MOV     WORD PTR DEV_INT+2,AX   ; STRATEGY
        MOV     AX,ES:[BX].SDEVSTRAT    ; & INTERRUPT
        MOV     WORD PTR DEV_STRAT,AX
        MOV     AX,ES:[BX].SDEVINT
        MOV     WORD PTR DEV_INT,AX
        POP     BX
        POP     ES                      ;GET EXISTING INT9
        MOV     WORD PTR OLD_INT9,BX
        MOV     BX,ES

        MOV     WORD PTR OLD_INT9+2,BX
        MOV     IOCTLB.IOS_CMD,IOC_ADS_CMD  ;SET FOR DEVICE
        MOV     REQFLW.RQI_NBYTES,TYPE IOS  ; STATUS IOCTL
        CALL    IOCTL           ;DO IOCTL TO GET STATUS
        OR      AX,AX           ; OF DRIVE
        JZ      OKFIOCTL
        MOV     DX,OFFSET BADFIOCTL
        MOV     BL,4            ;ERRORLEVEL4 - ERR DEV IOCTL
        JMP     ERROR_OUT
OKFIOCTL:
        TEST    WORD PTR IOCTLB.IOS_STAT,IOS_AUDIO_PLAY
        JNZ     OKDEVCD                 ;CHK PLAY SUPPORTED
        MOV     DX,OFFSET INSUFDD
        MOV     BL,5           ;ERRORLEVEL5-PLAY UNSUPPORTED
        JMP     ERROR_OUT
```

Listing 25.6 *continued*

```
OKDEVCD:
        CALL    TOC             ;READ THE TABLE OF CONTENTS
        MOV     IN_POPUP,1      ;DON'T ALLOW POPUP YET
        MOV     DX,OFFSET TRAP_KB
        PUSH    DS
        PUSH    CS
        POP     DS
        MOV     AX,2509H
        INT     21H             ;REPLACE INT9 ADDR VECTOR
        MOV     DX,OFFSET INSTALL_MSG
        MOV     AH,9
        INT     21H             ;SAY INSTALLED
        POP     DS
        MOV     ES,PSP
        MOV     AX,ES:[PSP_ENV]
        MOV     ES,AX           ;GET ENVIRONMENT ADDR
        MOV     AH,49H
        INT     21H             ;RELEASE ENVIRONMENT BLOCK
        MOV     DX,OFFSET KEEP_MEM
        ADD     DX,15
        MOV     CL,4
        SHR     DX,CL           ;CALC NUMBER OF PARAGRAPHS
        MOV     AX,CS           ; TO KEEP AFTER TSR
        ADD     DX,AX
        SUB     DX,PSP
        MOV     AX,3100H
        MOV     IN_POPUP,0      ;MAY NOW POPUP
        INT     21H             ;TERMINATE AND STAY RESIDENT
CDCNTL  ENDP
CODE    ENDS

STACKS  SEGMENT PARA STACK
        DW      256 DUP (?)
STACKS  ENDS
        END     CDCNTL

; End of File
```

Chapter 26

DMA Revealed

Karen Hazzah

DMA (Direct Memory Access) is a method for relieving the CPU from transferring data between memory and hardware devices, such as disk controllers and buffered communications devices. If you write device drivers or interfaces to hardware devices, you need a good understanding of DMA. This chapter explores the mysterious world of DMA in the PC environment: what it is, how it works on PC, AT, and Micro Channel systems, and how to write software to perform DMA in the DOS, Windows, and OS/2 environments.

What Is DMA?

The bus is the backbone of data movement in the PC. The bus connects the CPU to both memory and I/O devices. The CPU shares the bus with other devices that can transfer data to and from main memory. Such a device is called a "bus master" and when a bus master borrows the bus from the CPU to move data, that is called "cycle stealing." The PC's bus architecture prevents other bus masters from stealing more than one consecutive cycle from the CPU, so the CPU can continue processing.

Also, the CPU may not use the bus at all during some cycles, due to the presence of an instruction cache. So, the bus master idea is really an effective way of hardware multitasking, allowing other devices to share the bus with the CPU.

A program begins a DMA transfer by first programming the DMA controller, then programming the I/O device for the transfer. At this point, the CPU has completed its task, the DMA controller takes over, and the system begins a "DMA bus cycle." Although you must first program the DMA controller and perform whatever initialization the I/O device requires, the I/O device always initiates the actual data transfer by sending a signal to the DMA controller. In response to this signal, the DMA controller sends a signal to the CPU to indicate that the DMA controller will be bus master during the next cycle. When the CPU surrenders the bus, the DMA controller in turn acknowledges the original DMA request from the I/O device. At that point, the actual transfer of data can begin.

Exactly what happens next depends on the direction of the transfer. If the transfer is from memory to the I/O device, the DMA controller generates address and control signals to move data from the memory location onto the bus, and then generates I/O signals to notify the device that data is available on the bus. If the transfer is from the I/O device to memory, the DMA controller produces I/O signals that tell the device to move the data onto the bus, and then generates address and control signals to move data from the bus to the memory location.

At this point, one DMA bus cycle is complete, and the DMA controller increments (or decrements, depending on how you programmed it) its address register so that the next DMA bus cycle uses the next memory location. The DMA controller also decrements its counter at this time to determine whether or not the entire transfer is finished. If the counter is non-zero before the decrement, the DMA controller does nothing until the device once again signals that it is ready for DMA. When this happens, another DMA bus cycle begins. When the counter eventually changes from `0000h` to `FFFFh`, the DMA controller recognizes that the entire transfer is complete, and generates a special "Transfer Count Reached" signal on the bus. Most DMA devices use this signal to recognize the transfer is over and generate an interrupt to notify the CPU.

Which Bus?

The specifics of programming the DMA controller depend upon what type of bus it is connected to. The two main PC bus architectures are the AT bus (also called ISA) and the Micro Channel bus. However, programming the DMA requires you to further distinguish the 16-bit (true AT) bus from the 8-bit (original PC/XT) bus.

You cannot program the DMA controller for a 16-bit transfer on an 8-bit ISA bus, since the bus can only hold 8-bits of data at a time. Whether or not you can program a 16-bit transfer on a 16-bit ISA bus depends on whether the I/O device participating in the transfer is an 8-bit device or a 16-bit device. Because the 16-bit AT bus is backwardly- compatible with the 8-bit XT bus, many device adapters in 16-bit computers are actually 8-bit devices, and support only 8-bit DMA transfers. The converse is never true, however: you cannot install a 16-bit device adapter in an 8-bit computer.

Eight-Bit ISA DMAs

ISA DMA controllers support multiple DMA "channels." A DMA channel consists of a pair of acknowledge/request lines assigned to a unique device. An I/O device that supports DMA transfers must communicate with the DMA controller via a DMA channel that no other device is using. Typically, the I/O device is either hardwired for a particular channel or contains a DIP switch that you can set when you install the card to specify which DMA channel the device should use. Then, when you want to perform a DMA transfer with that device, you select the corresponding channel when you program the DMA controller.

The Intel 8237 DMA controller in the PC/XT supports four DMA channels. However, the PC/XT uses channel 0 for dynamic RAM refresh and channel 2 for the floppy disk controller. The acknowledge/request lines for these channels are not available on the PC expansion bus, leaving only channels 1 and 3 for use by 8-bit device adapters. Also, since the hard disk (standard on an XT, optional on a PC) uses DMA channel 3, it is likely that channel 1 is the only free channel for a DMA device in a PC/XT system.

It is quite possible to program the DMA controller to start a transfer on one channel before a transfer on another channel is finished. The DMA controller considers the channel with the lowest number to have the highest priority. So if a transfer is taking place on channel 3 and you program the DMA controller to start a transfer on channel 0, the controller suspends the transfer on channel 3 while it completes the transfer requested for channel 0. Only after the channel 0 transfer is complete will the DMA controller finish the transfer programmed for channel 3. This all takes place at the hardware level and is transparent to your software.

Address (hex)	**Description**
00	Base Address, Channel 0
01	Transfer Count, Channel 0
02	Base Address, Channel 1
03	Transfer Count, Channel 1
04	Base Address, Channel 2
05	Transfer Count, Channel 2
06	Base Address, Channel 3
07	Transfer Count, Channel 3
08	Status Register, Channels 0-3
0A	Mask Register, Channels 0-3
0B	Mode Register, Channels 0-3
0C	Byte Flip-Flop
0D	Controller Reset
0E	Clear Mask Register, Channels 0-3
0F	Write Mask Register, Channels 0-3
81	Page Table Address, Channel 2
82	Page Table Address, Channel 3
83	Page Table Address, Channel 1
87	Page Table Address, Channel 0

Table 26.1 ***DMA controller registers for channels 0-3.***

You program the Intel 8237 DMA controller by writing to several I/O registers. Table 26.1 lists these registers along with their I/O addresses. The mask register enables and disables a specific channel, which is necessary when you initialize the controller. A single mask register controls all four channels; a bit pattern in the register determines which channel is affected. A single mode register, which works in the same manner, determines the characteristics of a transfer, such as whether a transfer is a write (to memory) or a read (from memory), and whether the DMA controller should increment or decrement the address register during the transfer.

The controller contains a separate base address register for each channel. The base address register contains the physical address of the starting memory location for the DMA transfer. (I will discuss the details of determining this physical address later in this article.) The controller also contains a separate count register for each channel; this register contains the number of bytes to be transferred. However, you have to give the DMA controller a count one less than the total number of bytes, since the DMA controller considers the transfer finished when the count register changes from `0000h` to `FFFFh`.

Although the address register in the 8237 DMA controller holds only 16 bits, DMA transfers in a PC/XT system can actually take place anywhere in the entire 1Mb region addressed by the 8086 processor. The PC contains 4-bit page registers (one for each of the four DMA channels) that it combines with the DMA controller's address to form a complete 20-bit address. In addition to setting the 16-bit base address register in the DMA controller, you have to set the 4-bit page register corresponding to the DMA channel you want to use before the transfer. The I/O addresses for these page registers appear in Table 26.1.

Sixteen-Bit ISA DMAs

As I noted previously, the PC/XT effectively offers only one free DMA channel for I/O devices. The AT bus design overcame that limitation by incorporating two Intel 8237 DMA controllers, resulting in a total of eight possible DMA channels. Channel 2 is used by the floppy disk and channel

4 is used to connect the two controllers, leaving the rest of the channels (0,1,3,5,6,7) available for use by device adapters. Table 26.2 shows the I/O addresses of the registers in the additional 8237 DMA controller, which is used for channels 4 through 7.

The addresses of the original controller, used for channels 0 through 3, remain the same as in the original PC design, so you program DMA transfers with channels 0, 1, 2, or 3 exactly the same as for an 8-bit XT transfer. However, all transfers using channels 5, 6, or 7 must be 16-bit transfers. The additional DMA controller that handles channels 5-7 does not have a special "16-bit mode" to achieve these transfers, but 16-bit transfers do require special programming and one hardware trick.

Address (hex)	Description
89	Page Table Address, Channel 6
8A	Page Table Address, Channel 7
8B	Page Table Address, Channel 5
8F	Page Table Address, Channel 4
C0	Base Address, Channel 4
C2	Transfer Count, Channel 4
C4	Base Address, Channel 5
C6	Transfer Count, Channel 5
C8	Base Address, Channel 6
CA	Transfer Count, Channel 6
CC	Base Address, Channel 7
CE	Transfer Count, Channel 7
D0	Status Register, Channels 4-7
D4	Mask Register, Channels 4-7
D6	Mode Register, Channels 4-7
D8	Byte Flip-Flop
DA	Controller Reset
DC	Clear Mask Register, Channels 4-7
DE	Write Mask Register, Channels 4-7

Table 26.2 ***DMA controller registers for channels 4-7.***

Figure 26.1 illustrates how word transfers operate. This example demonstrates a DMA transfer of four bytes beginning at physical address `00002h`. To accomplish this, you first set the transfer count not to four, but to two (four shifted right one bit), because you need only half as many transfers when the controller is transferring twice as much data with each DMA cycle. You set the DMA controller base address register not to `000002H`, but to `000001H`, which is the original address shifted right by one bit.

This "shift magic" works because the address lines of the DMA controller (`A15` to `A0`), which correspond to the bits in the base address

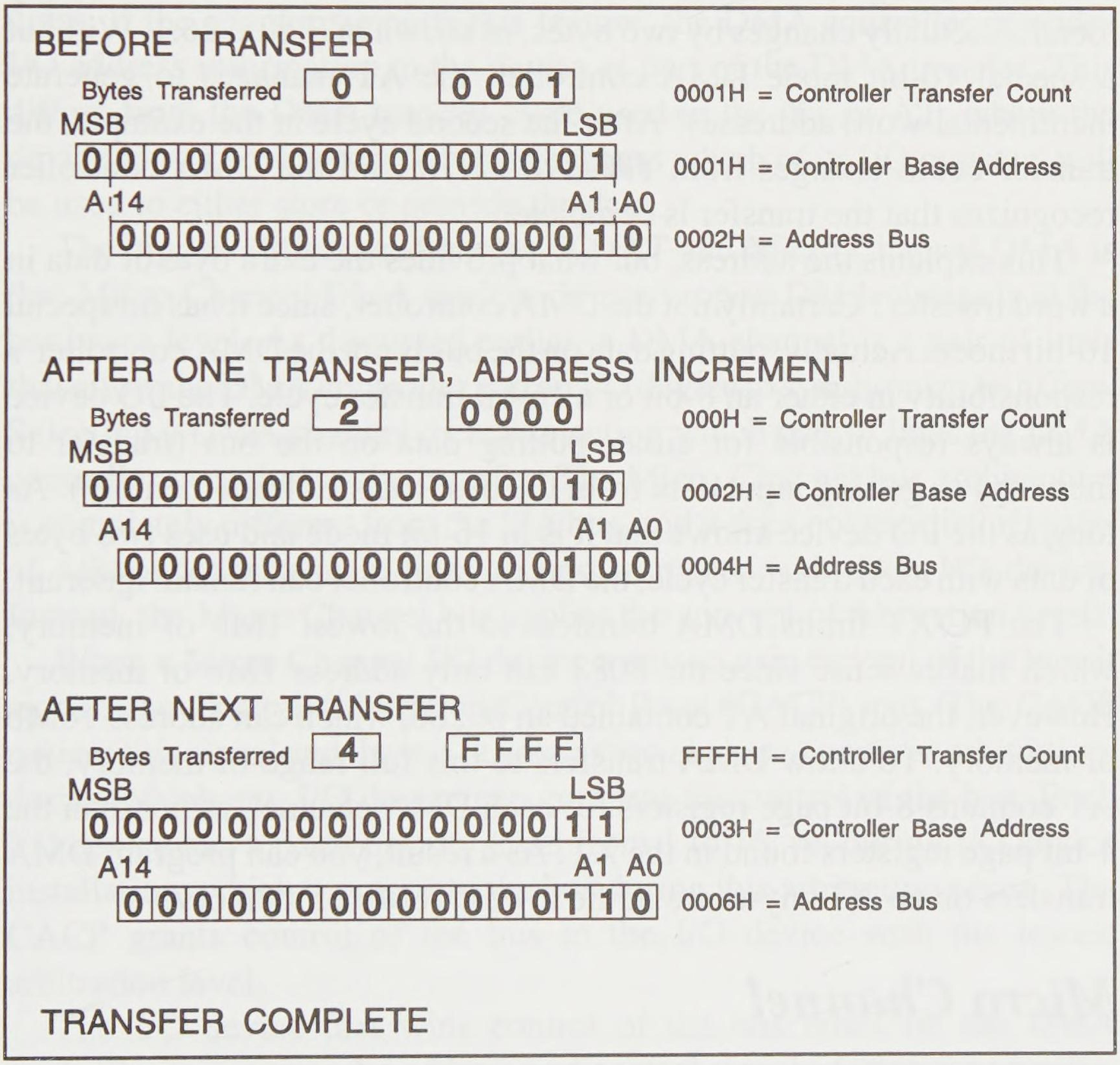

BEFORE TRANSFER
Bytes Transferred
0
0 0 0 1
0001H = Controller Transfer Count
MSB
LSB
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0001H = Controller Base Address
A14
A1
A0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0002H = Address Bus
AFTER ONE TRANSFER, ADDRESS INCREMENT
Bytes Transferred
2
0 0 0 0
000H = Controller Transfer Count
MSB
LSB
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0002H = Controller Base Address
A14
A1
A0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0004H = Address Bus
AFTER NEXT TRANSFER
Bytes Transferred
4
F F F F
FFFFH = Controller Transfer Count
MSB
LSB
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0003H = Controller Base Address
A14
A1
A0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0006H = Address Bus
TRANSFER COMPLETE

Figure 26.1 ***Illustration of the DMA controller incrementing its base address register at the end of each DMA cycle, changing the effective memory location by two bytes.***

register, are connected to the address bus in an unusual fashion. Output *A0*, the least significant bit of the base address register, is connected not to *A0* on the address bus, but to *A1*. Output *A1* is connected to *A2*, and so on. The result is that the address that you shifted right before programming the base address register is actually shifted left to produce the original address. This is simply a result of the way the controller's address outputs are connected to the bus.

Since AT architecture hardwires the second DMA controller's address lines "off-by-one," the DMA transfer address is effectively multiplied by two. Therefore, when the DMA controller increments (or decrements) its base address register at the end of each DMA cycle, the effective memory location actually changes by two bytes, as shown in Figure 26.1. Without a special 16-bit mode DMA controller, the AT manages to generate incremental word addresses! After the second cycle in the example, the transfer count changes from *0000h* to *FFFFh*, and the DMA controller recognizes that the transfer is complete.

This explains the address, but what provides the extra byte of data in a word transfer? Certainly not the DMA controller, since it has no special 16-bit mode. Actually, putting data on the bus is not the DMA controller's responsibility in either an 8-bit or a 16-bit transfer cycle. The I/O device is always responsible for either putting data on the bus (transfer to memory) or getting input data from the bus (transfer from memory). As long as the I/O device knows that it is in 16-bit mode and uses two bytes of data with each transfer cycle, the DMA controller can remain ignorant.

The PC/XT limits DMA transfers to the lowest 1Mb of memory, which makes sense since the 8088 can only address 1Mb of memory. However, the original AT contained an 80286, which can address 16Mb of memory. To allow DMA transfers to this full range of memory, the AT contains 8-bit page registers for each DMA channel, rather than the 4-bit page registers found in the XT. As a result, you can program DMA transfers on an AT anywhere in the lowest 16Mb of memory.

Micro Channel

Although device adapters for Micro Channel computers are completely different in hardware design than adapters for AT-bus computers, you can program the DMA controller in many cases in exactly the same manner as on an XT or AT computer. However, the Micro Channel DMA controller also includes a new interface to access the new features available to Micro Channel DMA devices. Also, even though the Micro Channel DMA controller provides an AT-style interface for your software, you should understand how it uses arbitration levels instead of DMA channels at the hardware level. This section discusses all of these issues.

One of the advanced features of the Micro Channel DMA controller is a 16-bit transfer size, available on all DMA channels. You must use the new interface to perform 16-bit transfers on channels 0 through 3, or to perform 8-bit transfers on channels 4 through 7, since these types of transfers are not AT-compatible.

Another Micro Channel-specific feature is a programmable I/O address. If the adapter supports this feature, the DMA controller provides I/O address information to the device as part of the DMA transfer. This differs from the DMA transfer cycle used in the XT or AT, where the device must know before the transfer starts which of its I/O registers will be used to either store or provide the data.

The biggest difference between XT/AT and Micro Channel DMA is that Micro Channel DMA devices do not support DMA channels at the hardware level. As discussed earlier, a DMA channel is a pair of lines that allow the DMA controller and an I/O device to synchronize transfers. Selecting a DMA channel means selecting which pair of lines the DMA controller uses during the transfer. The Micro Channel bus architecture is completely different from the ISA bus, and it does not use distinct pairs of acknowledge/request lines to communicate with each I/O device. Instead, the Micro Channel bus applies the concept of arbitration levels.

When a Micro Channel I/O device wants to gain control of the bus, it signals the Central Arbitration Control Point (CACP) unit. The CACP notices this signal and the bus begins a special phase known as arbitration, during which any I/O device can compete for control of the bus. Each I/O device has a fixed priority level from 1 to 15, set during I/O device installation, which it puts onto the bus during this arbitration phase. The CACP grants control of the bus to the I/O device with the lowest arbitration level.

The I/O device that wins control of the bus relies on the DMA controller to generate the address and control signals during the transfer. The Micro Channel DMA controller monitors the arbitration phase and compares the winning arbitration level with the list of arbitration levels it has been programmed to service. If it finds a match, the DMA controller becomes the bus master for this next bus cycle. Once the DMA controller becomes bus master, the bus cycle is similar to the ISA DMA transfer. At the end of the DMA bus cycle, the sequence begins again with another arbitration phase.

Micro Channel DMA devices are always described as being configured for a particular arbitration level, not a DMA channel. Unfortunately, documentation often mixes the terms "channel" and "arbitration level" when referring to the Micro Channel DMA controller. IBM's documentation for the controller refers to "programming the controller to service a DMA channel." It would be more accurate to say "service a DMA arbitration level," since the controller relies on arbitration levels to tell it when it will become bus master.

More confusion arises from the fact that you can configure a device for 15 possible arbitration levels, but the DMA controller supports only eight DMA channels. What, then, is the exact relationship between DMA channels and arbitration levels in a Micro Channel system? In most cases, the relationship is fixed: for example, DMA channel 1 always corresponds to arbitration level 1. That is, if you program the Micro Channel DMA controller for channel 1 (remember that the Micro Channel supports an AT-style software interface), it uses whatever device has arbitration level 1. DMA channels 1 through 3 correspond to arbitration levels 1 through 3 and DMA channels 5 through 7 correspond to arbitration levels 5 through 7.

Channels 0 and 4 are conspicuously missing from this list. These two channels, referred to as virtual channels, have a programmable arbitration level. To access a device configured to use an arbitration level greater than 7, you first program the Micro Channel DMA controller to associate a particular arbitration level with channel 0 or 4. This feature lets you use a single virtual DMA channel with more than one I/O device, so long as you use only one I/O device at a time with a specific virtual channel.

This alternating use of virtual channels is practical only in an environment where all software applications go through a common "DMA driver" to perform DMA transfers. This driver must keep track of which DMA channels are in use to prevent one application from reprogramming a virtual channel while another application's transfer is still in progress. Since only the OS/2 environment provides this insulating layer, through the Advanced BIOS (ABIOS) used by OS/2 device drivers, the virtual channel feature is practically useless under DOS and Windows.

To use any of these Micro Channel-specific features, you must use the controller's new interface, which is command-oriented. You execute a command by first writing a command byte to the controller's command register. The top four bits of the command byte specify the command, while the lower four bits specify the channel. If the command requires parameters, you follow the command byte by a write to the controller's data register. Some commands return information that you can read from the controller's data register. Table 26.3 shows the commands and register addresses.

Real-Mode Addresses

When you write code that accesses memory, CPU logic transforms your address into a physical memory address. This transformation is simple in real mode and more complex in protected mode. The DMA controller,

Address (hex)	**Register**
18	Command (command + channel)
1A	Data
Command (hex)	**Function**
00	Set I/O Address
20	Set Base Address
30	Get Base Address
40	Set Transfer Count
50	Get Transfer Count
60	Get Status
70	Get/Set Mode
80	Get/Set Arbitration
90	Set Mask Bit
A0	Clear Mask Bit
D0	Clear All Mask Bits

Table 26.3 ***Micro Channel DMA Controller Registers and Commands.***

however, communicates directly with the bus, and the addresses you program the controller with do not go through the normal transformations supplied by the CPU — they are raw physical addresses. This section discusses the implications of CPU addressing in programming the DMA controller and the restrictions caused by the DMA page register scheme.

In real mode, the CPU forms 20-bit addresses by shifting a 16-bit segment register left by four bits and adding a 16-bit offset. As discussed earlier, the DMA controller base address register is only 16-bits wide, so you have to place the upper four bits of the desired address into a separate 4-bit page register. External logic ensures that these page registers place the high four bits of the address on the bus when a DMA transfer takes place.

Although the DMA controller automatically increments its address register during a transfer in order to provide sequential addresses on the bus, there is no actual connection between the page registers and the DMA controller. For this reason, the page register does not increment when the controller's address register rolls over from `FFFFh` to `0000h`.

A simple example illustrates why this is a problem. Suppose you want to perform a DMA transfer of `1000h` bytes to the physical address `6F000h`. You store the lower 16 bits (`F000h`) in the DMA controller's address register and the high four bits (`6h`) in the correct DMA controller page register. If you program the controller for a transfer count of `1000h` bytes, the physical address of the last byte transferred should be `70000h` (`6F000h + 01000h`). But, since the DMA controller does not increment on the last byte, the physical address generated for the last byte transferred will actually be `60000h`, the wrong memory location.

The solution to this problem is simple: do not start a DMA transfer that will cross a 64Kb physical address boundary. If the ending physical address is on a different 64Kb page than the beginning physical address, you must use two separate DMA transfers, using the beginning page for the first transfer and the ending page for the second transfer.

Protected Mode

The Intel 80286, 80386 and 80486 also have another operating mode, called protected mode, which is used by operating systems like OS/2 and Windows. When operating in protected mode, the 80286 uses a 24-bit physical address; both the 80386 and 80486 use a 32-bit physical address.

Protected mode still utilizes segments and offsets, but the translation of logical (`segment:offset`) addresses into physical addresses is no longer a fixed mathematical calculation performed directly by the CPU. In protected mode, the CPU and the operating system together translate a logical address into a physical address. To program the DMA controller, you must therefore ask the operating system for the physical address of the buffer. Once you obtain the physical address, programming the DMA controller is the same regardless of CPU mode. Thus, the 64Kb page boundary rule discussed before still applies in protected mode.

Allocating the Buffer

A memory buffer that you want to use in a DMA transfer must meet several requirements. First, it must be fixed in physical address space;

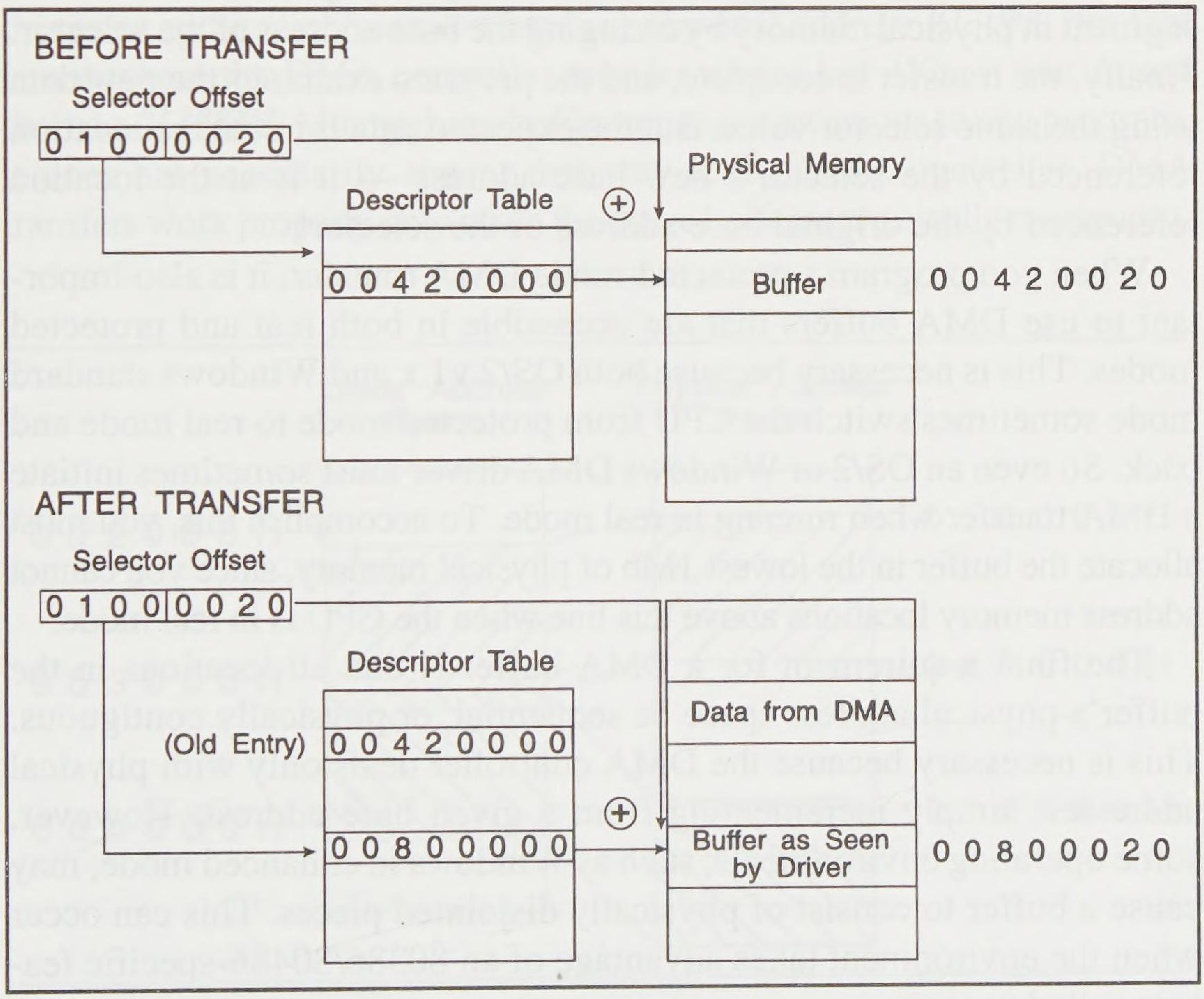

BEFORE TRANSFER
Selector Offset
0 1 0 0 0 0 2 0
Physical Memory
Descriptor Table
+
0 0 4 2 0 0 0 0
Buffer
0 0 4 2 0 0 2 0
AFTER TRANSFER
Selector Offset
0 1 0 0 0 0 2 0
Descriptor Table
Data from DMA
(Old Entry)
0 0 4 2 0 0 0 0
+
0 0 8 0 0 0 0 0
Buffer as Seen by Driver
0 0 8 0 0 0 2 0

Figure 26.2 *The results of performaing a DMA transfer to a moveable segment.*

that is, a given logical address (`segment:offset`) must always map to the same physical address. This is never a problem under DOS because, in real mode, the CPU always performs the same calculation to transform a given segment and offset logical address into a physical address. However, in protected-mode operating systems like OS/2 or Windows, memory can be either fixed in physical memory or moveable. In protected mode, a segment register is actually a selector — an indirect reference to the real base address. The operating system can move an application's segment in physical memory simply by changing the segment's base address. Since applications typically deal only with selector values and not physical addresses, applications are usually unaffected by the move. However, performing a DMA transfer to a moveable segment can have disastrous results.

Figure 26.2 illustrates what can go wrong. In this illustration, a program first sets up the DMA controller with the physical address associated with a given segment. Next, the operating system moves the segment in physical memory by changing the base address of the selector. Finally, the transfer is complete, and the program examines the new data using the same selector value. But the expected data is not at the location referenced by the selector's new base address — it is at the location referenced by the original base address of the selector!

When you program a protected-mode DMA transfer, it is also important to use DMA buffers that are accessible in both real and protected modes. This is necessary because both OS/2 v1.x and Windows standard mode sometimes switch the CPU from protected mode to real mode and back. So even an OS/2 or Windows DMA driver must sometimes initiate a DMA transfer when running in real mode. To accomplish this, you must allocate the buffer in the lowest 1Mb of physical memory, since you cannot address memory locations above this line when the CPU is in real mode.

The final requirement for a DMA buffer is that all locations in the buffer's physical address space be sequential, or physically contiguous. This is necessary because the DMA controller deals only with physical addresses, simply incrementing from a given base address. However, some operating environments, such as Windows in enhanced mode, may cause a buffer to consist of physically disjointed pieces. This can occur when the environment takes advantage of an 80386/80486-specific feature called paging.

Paging uses a second level of address translation in addition to segmentation. The chip first translates logical addresses to linear addresses and then to physical addresses. Each 4Kb page of linear address space can be mapped to a different location in physical memory, as illustrated in Figure 26.3. In this example, an 8Kb buffer located at logical address `0302:0000` is translated to linear address `002000H`. The entire 8Kb buffer actually consists of two pages in linear memory, the first at `002000H` and the second at `003000H`. In this example, the first page is further translated to physical address `010000H`, and the second page is translated to physical address `012000H`.

A DMA transfer of 8Kb (`2000H`) bytes from a device to this example buffer would not have the desired effect. The DMA driver would program the address register with the starting physical address `010000H`, and the transfer would fill in bytes `010000H` through `011FFFH`. However, look what happens when the DMA driver or application later tries to reference the second half (page) of this buffer, at linear address `003000H`. What the driver or application sees is what is at physical address `013000`. But that is not where the DMA controller transferred the last 4Kb: it transferred them to `011000H`! Although the buffer appears contiguous to any program, it does not necessarily appear that way to the DMA controller. DMA transfers work properly only when the entire buffer is physically contiguous.

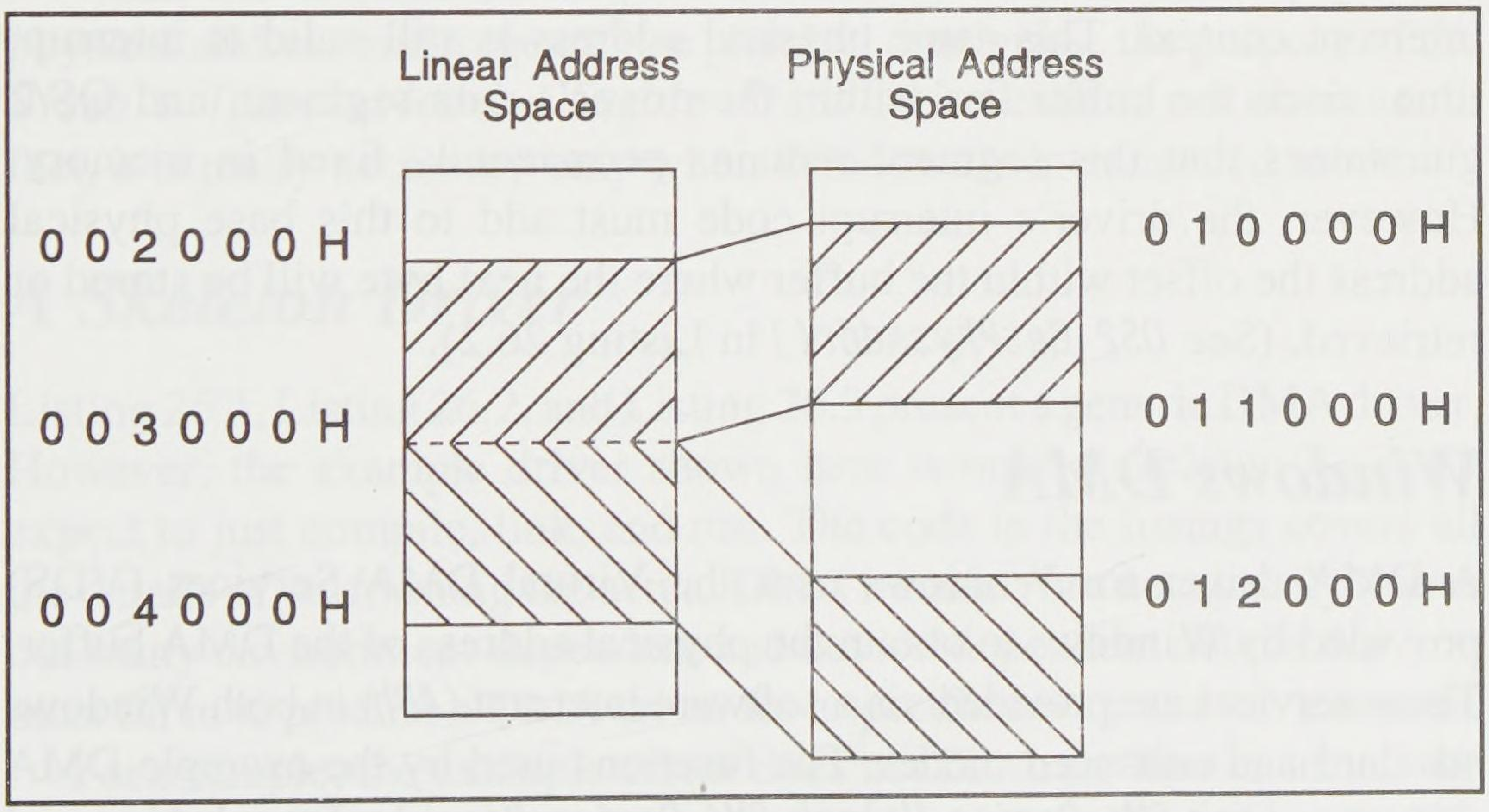


Figure 26.3 ***Example use of paging.***

How can a driver allocate a DMA buffer that meets all these requirements? The DMA buffer used by the example driver in the listings is declared as an array in the driver's data segment, and statically allocated by the compiler. This static allocation strategy meets the above requirements under DOS, Windows, and OS/2.

This is automatically true under DOS because all memory meets these requirements in real mode. Under OS/2, a device driver's data segment is always both fixed in physical memory and addressable in real mode. Since OS/2 does not use paging, all memory in a segment is physically contiguous. Under Windows, a DMA driver can guarantee the driver's data segment is both fixed and real mode addressable by building the driver as a DLL and declaring the data segment `FIXED` in the module definition files. However, the only way to guarantee that all pages of the segment are physically contiguous is to use the VDS Lock DMA Region service at runtime (discussed later in this article).

OS/2

An OS/2 DMA driver determines the physical address of its DMA buffer in two steps. At initialization time, the driver uses the `DevHelp VirtToPhys` service to convert the buffer's logical address (`segment:offset`) to a physical address. The driver must store this physical address for use at interrupt time, since the `VirtToPhys` service is not available in the interrupt context. This same physical address is still valid at interrupt time, since the buffer lies within the driver's data segment, and OS/2 guarantees that this segment remains permanently fixed in memory. However, the driver's interrupt code must add to this base physical address the offset within the buffer where the next byte will be stored or retrieved. (See `OS2_GetPhysAddr()` in Listing 26.2).

Windows DMA

A DMA driver for Windows uses the Virtual DMA Services (VDS) provided by Windows to obtain the physical address of the DMA buffer. These services are provided via a software interrupt (`4Bh`) in both Windows standard and enhanced modes. The functions used by the example DMA driver are `Lock DMA Region`, `Unlock DMA Region`, `Disable Translation`, and `Enable Translation`. For the complete specification, you should obtain the VDS documentation. This is available on the Microsoft BBS. Call 206-936-6735. The filename is *PW0519.EXE*. This section will describe how to use the essential VDS services.

To use the Lock DMA Region function, the driver first initializes a DMA descriptor structure with the selector and offset of the DMA buffer. Then the driver does an *INT 4Bh*, with *ES:DI* pointing to the DMA descriptor. If the requested region is physically contiguous and is thus suitable for DMA, the VDS function returns with the buffer's physical address in the DMA descriptor. If Windows is running in enhanced mode, it fixes each page of this region in physical memory until the driver calls the VDS *Unlock* service.

If the requested region is not physically contiguous, the VDS function returns the physical address of another memory region that is contiguous for the driver to use. If the DMA transfer is a read from memory then data will be copied from the driver's DMA buffer to the new buffer allocated by the VDS.

The allocated region is identified by a buffer ID in the DMA descriptor. When the DMA transfer is complete, the driver calls *Unlock* using the same DMA descriptor. During the *Unlock*, any allocated buffer is identified by the buffer ID and released. Also, if the DMA transfer was a write to memory from the I/O device, the *Unlock* copies the new data from the VDS allocated buffer to the driver's buffer.

While the *Lock/Unlock* sequence is clearly needed both to provide a physical address and to guarantee physical contiguity, the purpose of the *Disable Translation* and *Enable Translation* calls is not obvious. In fact, it is really necessary only when Windows is in enhanced mode.

A Skeleton Driver

Listing 26.1, Listing 26.2, and Listing 26.3 present a generic DMA driver. However, the example driver shown here is only a skeleton — don't expect to just compile, link, and run. The code in the listings covers all the details of performing the actual DMA transfer, but necessarily leaves out many environment-dependent and device-dependent details that you must fill in to produce a DMA driver for a specific device and environment.

For example, the example driver does not illustrate how to install the interrupt handler into the interrupt vector table, or exactly how and when to signal an *End Of Interrupt* to the interrupt controller during the interrupt service routine. The correct method for these steps depends on whether the driver is for a DOS, Windows, or OS/2 environment.

Also, the interrupt handler function makes many calls to device-specific functions, such as `Device_TypeOfInterrupt()`. These functions are identified by the `Device_` prefix in the function name. Environment-specific functions are similarly identified by the prefix `Env_`. You will not find the bodies of these functions in the listings because their implementation depends entirely on the device the driver is handling.

With these limitations in mind, you can examine the basic structure of the generic DMA driver. First of all, it is interrupt-driven: when the device is ready for a DMA transfer, it generates an interrupt that the driver's interrupt routine handles by initiating the DMA transfer. The device also generates another interrupt when the DMA transfer is complete. It is possible that a very simple DMA device may not have this capability to interrupt on completion. If not, you must make changes in the driver.

The other functional piece of the driver is the non-interrupt portion, which handles requests from the application to send or receive data. In

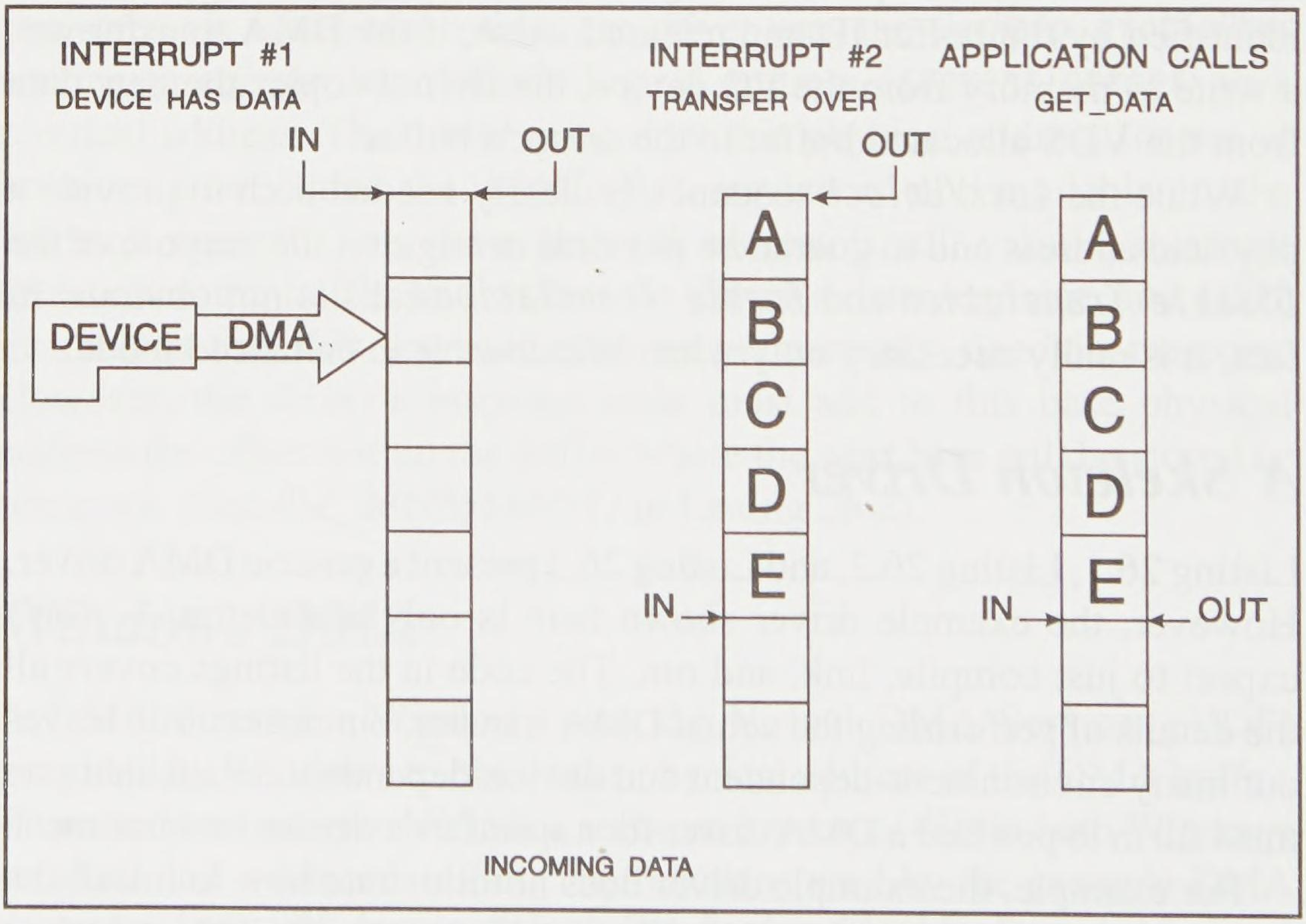


Figure 26.4 *Illustration of the device driver handling incoming data.*

this driver, these entry points are called `Put_Data()` and `Get_Data()`. These functions communicate with the interrupt routine via two different circular buffers and the pointers associated with the buffers.

Figure 26.4 shows how the driver handles incoming data. The driver's circular receive buffer is initially empty. When the device signals via an interrupt that it is ready for a transfer, the driver initiates a transfer from the device to the part of the receive buffer pointed to by the buffer's associated `IN` pointer. When the device signals that the transfer is complete, the driver increments the `IN` pointer by the number of bytes transferred. Later, when the application calls the driver's `Get_Data()` entry point, the driver copies data to the application's buffer from the driver's receive buffer, starting at the position indicated by the `OUT` pointer.

The driver processes outgoing data in a similar and symmetrical manner, as shown in Figure 26.5. The driver's circular transmit buffer is initially empty. When the application calls the driver's `Put_Data()` entry point, the driver copies from the application's data area to the part of the transmit buffer pointed to by the buffer's associated `IN` pointer. The driver then configures the device to generate an interrupt when the device is

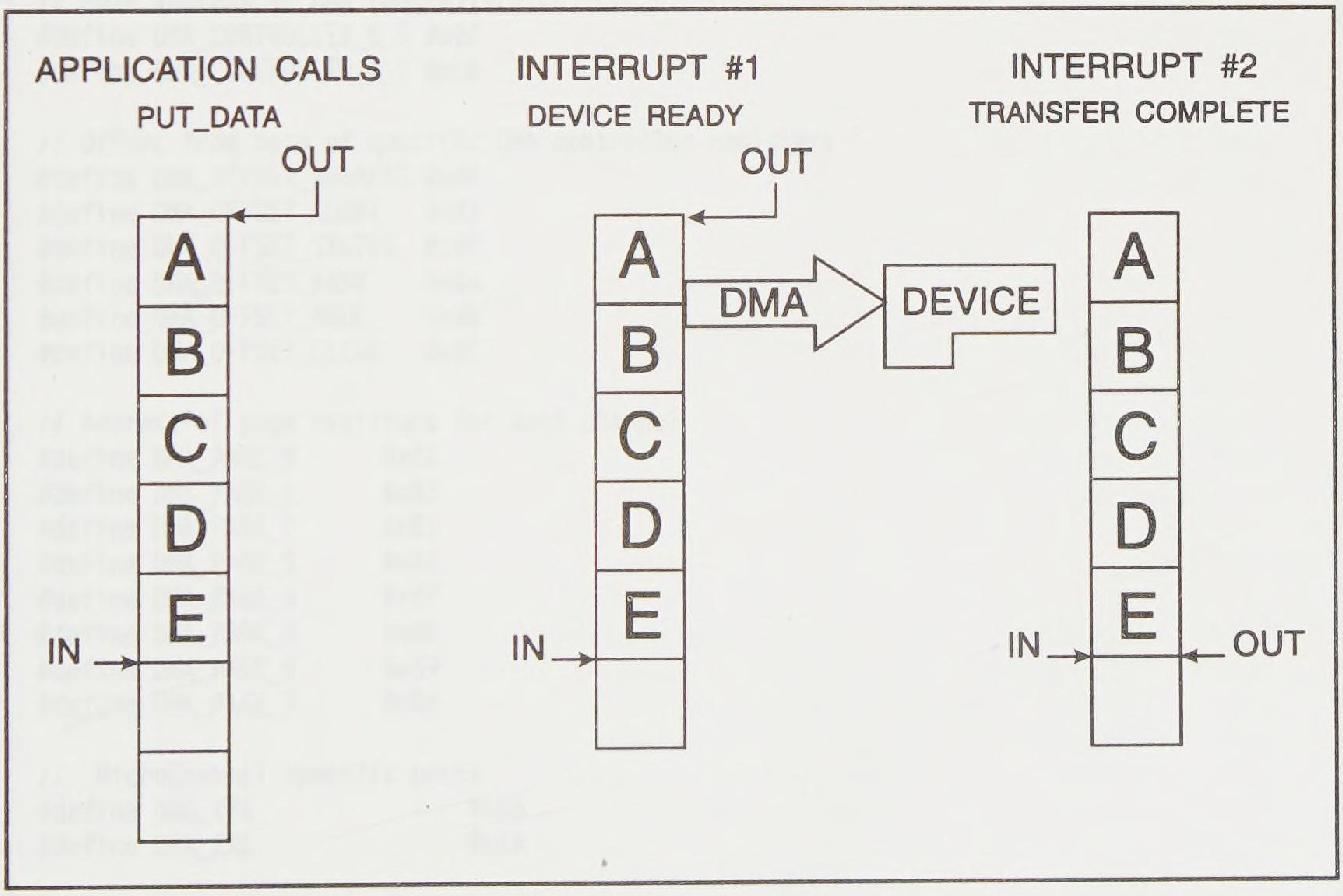


Figure 26.5 ***Illustration of the device driver handling outgoing data.***

ready to receive the data. When the interrupt occurs, the driver initiates a transfer from the part of the transmit buffer pointed to by the buffer's associated *OUT* pointer. When the device signals the transfer is complete, the driver increments the *OUT* pointer by the number of bytes transferred.

Summary

The DMA controller is one of the lesser known members of the PC architecture. Before you start writing your first device driver for a DMA device, you will want to have a solid understanding of the DMA controller, the I/O device your driver services, and the operating system your code will run under. This article and the skeleton generic DMA driver should put you on your way to your first successful DMA transfer.

Listing 26.1 `dma.h` — *DMA driver constants.*

```
typedef unsigned char BYTE;
typedef unsigned int  USHORT;
typedef unsigned long ULONG;

#define LOBYTE(w)      ((BYTE)(w))
#define HIBYTE(w)      ((BYTE)(((USHORT)(w) >> 8) & 0xff))
#define LOUSHORT(l)    ((USHORT)(l))
#define HIUSHORT(l)    ((USHORT)(((ULONG)(l) >> 16) & 0xffff))
#define MAKEULONG(l, h) ((ULONG)(((USHORT)(l)) | ((ULONG)((USHORT)(h))) << 16))

#define VDS_SERVICES             0x81
#define VDS_LOCK                 0x03
#define VDS_UNLOCK               0x04
#define VDS_ENABLE_TRANSLATION   0x0B
#define VDS_DISABLE_TRANSLATION  0x0C
#define VDS_FLAGS_COPY           0x02
#define VDS_FLAGS_ALIGN64K       0x10
#define VDS_FLAGS_ALIGN128K      0x20

// DMA controller addresses and commands
#define DMA_type_write  0x04
#define DMA_type_read   0x08
#define Transfer_Data   0x04
#define Write_Mem       0x08

// Base address of DMA controllers
#define DMA_CONTROLLER_0_3 0x00
#define DMA_CONTROLLER_4_7 0xC0

// Offset from base of specific DMA controller registers
#define DMA_OFFSET_ADDRESS 0x00
#define DMA_OFFSET_COUNT   0x01
#define DMA_OFFSET_STATUS  0x08
#define DMA_OFFSET_MASK    0x0A
#define DMA_OFFSET_MODE    0x0B
#define DMA_OFFSET_CLEAR   0x0C

// Address of page registers for each channel
#define DMA_PAGE_0      0x87
#define DMA_PAGE_1      0x83
#define DMA_PAGE_2      0x81
#define DMA_PAGE_3      0x82
#define DMA_PAGE_4      0x8F
#define DMA_PAGE_5      0x8B
#define DMA_PAGE_6      0x89
#define DMA_PAGE_7      0x8A

//  MicroChannel specific ports
#define DMA_XFN             0x18
#define DMA_EXE             0x1A
```

Listing 26.1 *continued*

```
// MicroChannel extended commands
#define Set_IO_Adr              0x00
#define Set_Mem_Adr             0x20
#define Get_Mem_Adr             0x30
#define Set_Count               0x40
#define Get_Count               0x50
#define Get_Status              0x60
#define Set_Mode                0x70
#define Set_Arbus               0x80
#define Set_Chn_Mask            0x90
#define Reset_Chn_Mask          0xA0
#define Master_Clear            0xD0

#define BUS_MICROCHANNEL    01
#define BUS_AT_ISA          02

#ifdef DOS
#define GET_PHYS_ADDR( pchar, dmastruc, bufstruc ) Dos_GetPhysAddr( pchar, dmastruc )
#endif
#ifdef WIN3
#define GET_PHYS_ADDR( pchar, dmastruc, bufstruc ) Win3_GetPhysAddr( pchar, dmastruc )
#endif
#ifdef OS2
#define GET_PHYS_ADDR( pchar, dmastruc, bufstruc ) OS2_GetPhysAddr( pchar, dmastruc, bufstruc )
#endif

typedef enum
{
  TRANSFER_READ_FROM_MEM,
  TRANSFER_WRITE_TO_MEM
} TRANSFER_TYPE;

typedef struct
{
   ULONG    Size;
   ULONG    Linear;
   USHORT   Selector;
   USHORT   BufferID;
   ULONG    Physical;
} DMA_DESCRIPTOR;

typedef struct
{
   ULONG    PhysAddr;      // used only by OS/2 drivers
   char     *pStart;       // first byte in buffer
   char     *pEnd;         // last byte in buffer
   char     *pIn;          // next byte to be added
   char     *pOut;         // next byte to be removed
   USHORT   Size;          // total size of buffer
} BUF_INFO;
```

Listing 26.1 *continued*

```
typedef struct
{
   BUF_INFO RxBuf;
   BUF_INFO TxBuf;
   BYTE     BusType;
   BYTE     Channel;
   TRANSFER_TYPE TransferType;    // read or write
   ULONG    TransferPhys;         // physical address of
                                  // buffer for DMA transfer
   USHORT   TransferCount;        // number of bytes for DMA transfer
   char **far *pVDSInfo;
} DMA_INFO;

typedef struct
{
  unsigned int   DeviceRdyToTx:1;
  unsigned int   DeviceRdyToRx:1;
  unsigned int   DmaComplete:1;
  unsigned int   DmaAbort:1;
} INTERRUPT_TYPE;

/* End of File */
```

Listing 26.2 `dmaapi.c` — *DMA driver API code.*

```
#include "dma.h"

//--- External References ----------------
extern void Dev_Start_Transmission( void );
extern ULONG pfnDevHlp;
extern DMA_INFO DmaInfo;   // in "DMAISR.C"

//--- Local Functions --------------------
void OS2_Call_During_Init( DMA_INFO * );
USHORT GetData(  char *, USHORT );
USHORT PutData(  char *, USHORT );

// This function MUST be called during driver
// initialization, NOT from the interrupt service routine.
void OS2_Call_During_Init( DMA_INFO *pDmaInfo )
{
#define DEVHLP_VIRT_TO_PHYS   0x16
   USHORT   HiWord;
   USHORT   LoWord;
   char     *pBuf;

   // Use VirtToPhys DevHelp function to determine physical
   // address of first byte of driver's receive buffer.
   pBuf = pDmaInfo->RxBuf.pStart;
   _asm
   {
      mov    di, pDmaInfo
      mov    si, pBuf
      mov    dl, 16h        ; command code for VirtToPhys
      call   pfnDevHlp      ; function pointer to kernel's DevHelp entry point
      mov    HiWord, ax     ; (AX) = high-order 16 bits of phys addr
      mov    LoWord, dx     ; (DX) = low-order 16 bits of phys addr
   }
   pDmaInfo->RxBuf.PhysAddr = MAKEULONG( LoWord, HiWord );
   // Use VirtToPhys DevHelp function to determine physical
   // address of first byte of driver's transmit buffer.
   pBuf = pDmaInfo->TxBuf.pStart;
   _asm
   {
      mov    di, pDmaInfo
      mov    si, pBuf
      mov    dl, DEVHLP_VIRT_TO_PHYS
      call   pfnDevHlp      ; function pointer to kernel's DevHelp entry point
      mov    HiWord, ax     ; (AX) = high-order 16 bits of phys addr
      mov    LoWord, dx     ; (DX) = low-order 16 bits of phys addr
   }
   pDmaInfo->TxBuf.PhysAddr = MAKEULONG( LoWord, HiWord );
}
```

Listing 26.2 *continued*

```
USHORT GetData(char *buffer,  //  Appl. data location
               USHORT length) //  Length to be copied
{
  char    *pOut, *pIn;
  USHORT   size, lengthx;

  // Derefence pointers for speed in loop.
  pOut = DmaInfo.RxBuf.pOut;
  pIn = DmaInfo.RxBuf.pIn;
  lengthx = 0;
  // Compute remaining free space in receive buffer
  size = pIn - pOut;
  if (size < 0)
    size += DmaInfo.RxBuf.Size;
  // If we can't give 'em all they want, give 'em all we got
  if (length > size)
    length = size;
  // Copy data in chunks until input data buffer filled
  while (length)    // buffer length not exhausted
  {
    if (pIn > pOut)
    {
      size = pIn - pOut;
      if (size > length)
        size = length;
      memcpy( buffer, pOut, size );
      pOut += size;
    }
    else if (pIn < pOut)
    {
      size = DmaInfo.RxBuf.pEnd - pOut + 1;
      if (size > length)
        size = length;
      memcpy( buffer, pOut, size );
      pOut += size;
      if (pOut > DmaInfo.RxBuf.pEnd)
        pOut = DmaInfo.RxBuf.pStart;
    }
    else     // data in our buffer exhausted
      break;

    length  -= size;
    lengthx += size;
    buffer  += size;
  }
  DmaInfo.RxBuf.pOut = pOut;
  return( lengthx );   // number of bytes actually copied
}
```

Listing 26.2 *continued*

```
USHORT PutData(char *buffer,  //  Appl. data location
               USHORT length) //  Length to be copied
{
   char      *pIn, *pOut;
   USHORT   size, lengthx;

   // Dereference pointers for speed in loop.
   pIn = DmaInfo.TxBuf.pIn;
   pOut = DmaInfo.TxBuf.pOut;
   lengthx = 0;
   // Compute remaining free space in transmit buffer
   size = pOut - pIn - 1;
   if (size < 0)
      size += DmaInfo.TxBuf.Size;
   if (length > size)
      length = size;
   while (length)
   {
      if (pIn < pOut)
      {
         size = pOut - pIn - 1;
         if (size > length)
            size = length;
         memcpy( pIn, buffer, size );
         pIn += size;
      }
      else
      {
         size = DmaInfo.TxBuf.pEnd - pIn + 1;
         if (size > length)
            size = length;
         memcpy( pIn, buffer, size );
         pIn += size;
         if (pIn > DmaInfo.TxBuf.pEnd)
         pIn = DmaInfo.TxBuf.pStart;
      }
      length  -= size;
      lengthx += size;
      buffer  += size;
   }
   DmaInfo.TxBuf.pIn = pIn;
   if (lengthx)
      Dev_StartTransmission();
   return( lengthx );
}

/* End of File */
```

Listing 26.3 `dmaisr.c` — *DMA driver interrupt code.*

```
#include <dos.h>
#include "dma.h"

//--- Global Data ----------------------------------------------------------
BYTE PageRegister[ 8 ] = { DMA_PAGE_0, DMA_PAGE_1, DMA_PAGE_2, DMA_PAGE_3,
                           DMA_PAGE_4, DMA_PAGE_5, DMA_PAGE_6, DMA_PAGE_7 };
DMA_DESCRIPTOR DmaDescriptor;
DMA_INFO DmaInfo;
//--- External References --------------------------------------------------
extern void Device_RetrieveDmaInfo( DMA_INFO * );
extern INTERRUPT_TYPE Device_TypeOfInterrupt( void );
extern USHORT Device_QuerySpaceAvail( void );
extern USHORT Device_QueryDataAvail( void );
extern Env_DoEoi( void );
//--- Local Functions ------------------------------------------------------
void DmaAbort(DMA_INFO *);
void DmaSendComplete( DMA_INFO * );
void DmaReceiveComplete( DMA_INFO * );
void DmaSend( DMA_INFO * );
void DmaReceive( DMA_INFO * );
void ProgramDmaController( DMA_INFO * );
USHORT ReadTransferCount( DMA_INFO * );
void CheckForPageWrap( USHORT *pCount, ULONG physAddr );
#ifdef DOS
ULONG Dos_GetPhysAddr( char far *pBuffer, DMA_INFO *pDmaInfo );
#endif
#ifdef WIN3
ULONG Win3_GetPhysAddr( char far * pBuffer, DMA_INFO *pDmaInfo );
void Win3_ReleaseBuffer( DMA_INFO * );
#endif
#ifdef OS2
ULONG OS2_GetPhysAddr( char *pBuffer, DMA_INFO *pDmaInfo, BUF_INFO *pBufStruc );
#endif
```

Listing 26.3 *continued*

```
void InterruptHandler( void )
{
   INTERRUPT_TYPE intType;

   /* Determine cause(s) of interrupt. */
   intType = Device_TypeOfInterrupt();

   /* Interrupt priority is:
    * 1. DmaAbort
    * 2. DmaComplete
    * 3. DeviceRdyToTx
    * 4. DeviceRdyToRx */
   if (intType.DmaAbort)
      DmaAbort( &DmaInfo );
   if (intType.DmaComplete)
   {
      /* Interrupt type doesn't tell us whether completed
       * transfer was TO device or FROM device. Use
       * TransferType field to decide. */
      if (DmaInfo.TransferType == TRANSFER_WRITE_TO_MEM )
         DmaReceiveComplete( &DmaInfo );
      else if (DmaInfo.TransferType == TRANSFER_READ_FROM_MEM)
         DmaSendComplete( &DmaInfo );
   }

  /* After DmaComplete housekeeping, driver is
   * ready to handle another DMA transfer during
   * same interrupt, if device is ready. */
  if (intType.DeviceRdyToTx)
     DmaReceive( &DmaInfo);
  if (intType.DeviceRdyToRx)
     DmaSend( &DmaInfo);
   /* DMA stuff is complete. Must send EOI to
    * interrupt controller before returning. */
   Env_DoEoi();
}
```

Listing 26.3 *continued*

```
void DmaAbort( DMA_INFO *pDmaInfo )
{
   USHORT   bytesNotTransferred;

/* Must determine how many bytes were transferred
 * before abort. First ask DMA controller how many
 * bytes have not yet been transferred. Then
 * subtract from original transfer count. */
   bytesNotTransferred = ReadTransferCount( pDmaInfo );
   pDmaInfo->TransferCount = pDmaInfo->TransferCount - bytesNotTransferred;
   // Now that TransferCount field is updated,
   // call subroutine to advance queue pointers.
   if (pDmaInfo->TransferType == TRANSFER_WRITE_TO_MEM)
      DmaReceiveComplete( pDmaInfo );
   else if (pDmaInfo->TransferType == TRANSFER_READ_FROM_MEM)
      DmaSendComplete( pDmaInfo );
}

void DmaReceiveComplete( DMA_INFO *pDmaInfo )
{
   // Update receive queue ptr, accounting for wrap.
   pDmaInfo->RxBuf.pIn += pDmaInfo->TransferCount;
   if (pDmaInfo->RxBuf.pIn > pDmaInfo->RxBuf.pEnd)
      pDmaInfo->RxBuf.pIn = pDmaInfo->RxBuf.pStart;

#ifdef WIN3
   Win3_ReleaseBuffer( pDmaInfo );
#endif

}

void DmaSendComplete( DMA_INFO *pDmaInfo )
{
   // Update transmit queue ptr, accounting for wrap.
   pDmaInfo->TxBuf.pOut += pDmaInfo->TransferCount;
   if (pDmaInfo->TxBuf.pOut > pDmaInfo->TxBuf.pEnd)
      pDmaInfo->TxBuf.pOut = pDmaInfo->TxBuf.pStart;

#ifdef WIN3
   Win3_ReleaseBuffer( pDmaInfo );
#endif

}
```

Listing 26.3 *continued*

```
void DmaSend( DMA_INFO *pDmaInfo )
{
   char      *pIn, *pOut;
   USHORT    bytesInQueue;
   USHORT    roomInDevice;

   // Calculate bytes in driver's transmit queue.
   pOut = pDmaInfo->TxBuf.pOut;
   pIn = pDmaInfo->TxBuf.pIn;
   if (pIn >= pOut)
   {
      bytesInQueue = pIn - pOut;
   }
   else
   {
      // DMA transfer data must be contiguous, so stop at end of buffer.
      bytesInQueue = pDmaInfo->TxBuf.pEnd - pOut + 1;
   }

   // Ask device how much space it has available for new data.
   // Device may not have enough space for all we have.
   roomInDevice = Dev_QuerySpaceAvail();
   if (roomInDevice < bytesInQueue)
      pDmaInfo->TransferCount = roomInDevice;
   else
      pDmaInfo->TransferCount = bytesInQueue;

   // Determine physical address of first byte in DMA
   // transfer. Since we're sending data _to_ device,
   // use the pOut pointer.
   pDmaInfo->TransferPhys =
      GET_PHYS_ADDR( (char far *)pDmaInfo->TxBuf.pOut, pDmaInfo, &pDmaInfo->TxBuf );

   pDmaInfo->TransferType = TRANSFER_READ_FROM_MEM;
   ProgramDmaController( pDmaInfo );
}
```

Listing 26.3 *continued*

```
void DmaReceive( DMA_INFO *pDmaInfo )
{
   USHORT   spaceAvail;
   USHORT   dataAvail;
   char     *pOut, *pIn;

   // Calculate space in driver's receive queue.
   pIn = pDmaInfo->RxBuf.pIn;
   pOut = pDmaInfo->RxBuf.pOut;
   if (pOut > pIn)
   {
      spaceAvail = pOut - pIn - 1;
   }
   else
   {
      // DMA transfer data must be contiguous,
      // so stop at end of buffer.
      spaceAvail = pDmaInfo->RxBuf.pEnd - pIn + 1;
      if (pOut == pDmaInfo->RxBuf.pStart)
         spaceAvail--;  // must leave last byte empty
   }
   if (spaceAvail)
   {
      dataAvail = Dev_QueryDataAvail();
      if (dataAvail > spaceAvail)
         pDmaInfo->TransferCount = spaceAvail;
      else
         pDmaInfo->TransferCount = dataAvail;

      // Determine physical address of first byte in
      // DMA transfer. Since we're receiving data
      // _from_ device, use the pIn pointer.
      pDmaInfo->TransferPhys =
            GET_PHYS_ADDR( (char far *)pDmaInfo->RxBuf.pIn, pDmaInfo, &pDmaInfo->RxBuf );
      pDmaInfo->TransferType = TRANSFER_WRITE_TO_MEM;
      ProgramDmaController( pDmaInfo );
   }
}
```

Listing 26.3 *continued*

```
USHORT ReadTransferCount( DMA_INFO *pDmaInfo )
{
   USHORT   count;
   BYTE     regTransferCount;
   BYTE     channel = pDmaInfo->Channel;

   if (pDmaInfo->BusType != BUS_MICROCHANNEL)
   {
      if (channel <= 3)
      {
         regTransferCount = DMA_CONTROLLER_0_3 + DMA_OFFSET_COUNT;
         outp( DMA_CONTROLLER_0_3 + DMA_OFFSET_CLEAR, 0 );
      }
      else
      {
         regTransferCount = DMA_CONTROLLER_4_7 + DMA_OFFSET_COUNT;
         outp( DMA_CONTROLLER_4_7 + DMA_OFFSET_CLEAR, 0 );
      }
      regTransferCount += (channel << 1);
      // First byte read is LSB of count.
      count = inp( regTransferCount );
      // Second byte read is MSB of count. Combine with LSB.
      count = (count << 8) | (inp( regTransferCount ) );
   }
   else
   {
      outp( DMA_XFN, Get_Count + channel );
      // First byte read is LSB of count.
      count = inp( DMA_EXE );
      // Second byte read is MSB of count. Combine with LSB.
      count = (count >> 8) | inp( DMA_EXE );
   }
   return( count - 1 );
}
```

Listing 26.3 *continued*

```
void ProgramDmaController( DMA_INFO * pDmaInfo )
{
   ULONG     physAddr = pDmaInfo->TransferPhys;
   USHORT    count = pDmaInfo->TransferCount;
   BYTE      channel = pDmaInfo->Channel;
   BYTE      baseReg;
   BYTE      tempReg;

   // Controller chips on MicroChannel are different than XT or AT.
   if (pDmaInfo->BusType != BUS_MICROCHANNEL)
   {
      if (channel <= 3)
      {
         baseReg = DMA_CONTROLLER_0_3;
         outp( DMA_CONTROLLER_0_3 + DMA_OFFSET_CLEAR, 0 );
      }
      else
      {
         baseReg = DMA_CONTROLLER_4_7;
         outp( DMA_CONTROLLER_4_7 + DMA_OFFSET_CLEAR, 0 );
      }
      // Set channel's mask bit to disable channel
      outp( baseReg + DMA_OFFSET_MASK, channel | 0x04 );
      // Set mode to read or write.
      if (pDmaInfo->TransferType == TRANSFER_READ_FROM_MEM)
      {
         outp( baseReg + DMA_OFFSET_MODE, channel | DMA_type_read );
      }
      else
      {
         outp( baseReg + DMA_OFFSET_MODE, channel | DMA_type_write );
      }
      // Set up address register. Remember to do
      // "shift magic" if transfer will be a 16-bit transfer (channels 4-7).
      if (channel >= 4)
         physAddr >>= 1;
      tempReg = baseReg + DMA_OFFSET_ADDRESS + (channel << 1);
      outp( tempReg, LOBYTE((LUSHORT( physAddr ))) );  // bits 7-0  go first
      outp( tempReg, HIBYTE((LUSHORT( physAddr ))) );  // bits 15-8 go next
      // Set up transfer count, which is always I/O address after address register.
      tempReg++;
      count = pDmaInfo->TransferCount - 1;
      if (channel >= 4)
         count >>= 1;
      outp( tempReg, LOBYTE( count ) );   // bits 7-0
      outp( tempReg, HIBYTE( count ) );   // bits 15-8
      // Set up page table register.
      tempReg = PageRegister[ channel ];
      outp( tempReg, LOBYTE(HIUSHORT( physAddr )) );  // bits 23-16
      // Clear channel's mask bit to enable channel
      outp( baseReg + DMA_OFFSET_MASK, channel );
   }
```

Listing 26.3 *continued*

```
   else
   {
      // Disable transfers on this channel.
      outp( DMA_XFN, Set_Chn_Mask + channel );
      // Set channel mode to read or write.
      outp( DMA_XFN, Set_Mode + channel );
      if (pDmaInfo->TransferType == TRANSFER_READ_FROM_MEM)
      {
         outp( DMA_EXE, Transfer_Data );
      }
      else
      {
         outp( DMA_EXE, Transfer_Data | Write_Mem );
      }
      // Set up address register
      outp( DMA_XFN, channel + Set_Mem_Adr );
      outp( DMA_EXE, LOBYTE((LUSHORT( physAddr ))) );
      outp( DMA_EXE, HIBYTE((LUSHORT( physAddr ))) );
      outp( DMA_EXE, LOBYTE((HUSHORT( physAddr ))) );
      // Set up transfer count.
      count = pDmaInfo->TransferCount - 1;
      outp( DMA_XFN, Set_Count + channel );
      outp( DMA_EXE, LOBYTE( count ) ); // bits 7-0
      outp( DMA_EXE, HIBYTE( count ) ); // bits 15-8
      // Enable transfers on this channel.
      outp( DMA_XFN, Reset_Chn_Mask + channel );
   }
}

#ifdef DOS
ULONG Dos_GetPhysAddr( char far *pBuffer, DMA_INFO *pDmaInfo )
{
   ULONG       physAddr;

   physAddr = (ULONG)FP_SEG( pBuffer );
   physAddr <<= 4;
   physAddr += (ULONG)FP_OFF( pBuffer );
   // If transfer will cross over 64K page boundary, shorten transfer by updating count.
   CheckForPageWrap( &(pDmaInfo->TransferCount), physAddr );
   return( physAddr );
}
#endif
```

Listing 26.3 *continued*

```
#ifdef WIN3
ULONG Win3_GetPhysAddr( char far * pBuffer, DMA_INFO *pDmaInfo )
{
   DMA_DESCRIPTOR *pDmaDescriptor;
   char  far      *pVdsCheckByte;
   BYTE           channel = pDmaInfo->Channel;

   // Can't usually point directly to physical
   // memory in Windows, but Windows guarantees
   // selector 0x40 points to physical 0x0400.
   FP_SEG( pVdsCheckByte ) = 0x40;
   FP_OFF( pVdsCheckByte ) = 0x7B;
   if (!(*pVdsCheckByte & 0x20))
   {
      // VDS is not available, do real mode calculation.
      Dos_GetPhysAddr( pBuffer, pDmaInfo );
   }
   else
   {
      // Disable Translation
      _asm
      {
         mov   ah, VDS_SERVICES
         mov   al, VDS_DISABLE_TRANSLATION
         mov   bl, channel
         int   0x4B
      }

      // Lock buffer region
      DmaDescriptor.Selector = FP_SEG( pBuffer );
      DmaDescriptor.Linear = FP_OFF( pBuffer );
      DmaDescriptor.Size = pDmaInfo->TransferCount;
      pDmaDescriptor = &DmaDescriptor;
      _asm
      {
         mov   ah, VDS_SERVICES
         mov   al, VDS_LOCK
         mov   dx, VDS_FLAGS_COPY+VDS_FLAGS_ALIGN64K+VDS_FLAGS_ALIGN128K
         mov   di, pDmaDescriptor
         push  es
         push  ds
         pop   es
         int   0x4B
         pop   es
      }

      return( DmaDescriptor.Physical );
   }

}
```

Listing 26.3 *continued*

```
void Win3_ReleaseBuffer( DMA_INFO *pDmaInfo )
{
   BYTE    channel = pDmaInfo->Channel;
   DMA_DESCRIPTOR *pDmaDescriptor;

   // Enable Translation
   _asm
   {
      mov   ah, VDS_SERVICES
      mov   al, VDS_ENABLE_TRANSLATION
      mov   bl, channel
      int   0x4B
   }
   // Unlock buffer region
   pDmaDescriptor = &DmaDescriptor;
   _asm
   {
      mov   ah, VDS_SERVICES
      mov   al, VDS_UNLOCK
      mov   dx, VDS_FLAGS_COPY
      mov   di, pDmaDescriptor
      push  es
      push  ds
      pop   es
      int   0x4B
      pop   es
   }
}
#endif

#ifdef OS2
ULONG OS2_GetPhysAddr( char *pBuffer, DMA_INFO *pDmaInfo, BUF_INFO *pBufStruc )
{
   char  *pDistanceFromStart;
   ULONG physAddr;

   pDistanceFromStart = pBuffer - pBufStruc->pStart;
   physAddr = pBufStruc->PhysAddr + (ULONG)pDistanceFromStart;
   CheckForPageWrap( &(pDmaInfo->TransferCount), physAddr );
   return( physAddr );
}
#endif

void CheckForPageWrap( USHORT *pCount, ULONG physAddr )
{
   ULONG      endPhysAddr;

   endPhysAddr = physAddr + (ULONG)*pCount - 1;
   if ( (endPhysAddr & 0xFFFF0000) != (physAddr & 0xFFFF0000) )
      *pCount = 0x0000 - (USHORT)physAddr;
}

/* End of File */
```

Chapter 27

Self-Loading Device Drivers for DOS

Tomas Nelson

The introduction of installable device drivers in DOS v2.0 meant greatly increased flexibility for the control of peripheral devices. If you've ever tried to run a program that required the presence of a device driver, however, you know how inconvenient it can be to have to edit `config.sys` and reboot. A device driver capable of loading directly from the command line (or automatically from `autoexec.bat`) would offer a greater degree of freedom and convenience. This chapter explores one method of building a device driver with such capability. As a bonus, the technique described here permits a much easier way to test and debug device drivers.

How DOS Loads a Device Driver

The DOS BIOS module contains a number of resident device drivers that the bootstrap routine loads automatically during the boot sequence.

		Offsets		
Field	**Size**	**v2**	**v3.0**	**v3.1+**
Segment MCB Chain Head	WORD	-02h	-02h	-02h
1st Drive Param Block	DWORD	00h	00h	00h
DCB List Head	DWORD	04h	04h	04h
Clock$ Device Header	DWORD	08h	08h	08h
CON Device Header	DWORD	0Ch	0Ch	0Ch
# Logical Drives (1)	BYTE	10h	1Bh	21h
Max# BPS on Block Device	WORD	11h	11h	10h
Ptr 1st Disk Buffer (2)	DWORD	13h	13h	12h
Pointer CDS Table	DWORD	n/a	17h	16h
Pointer FCB Chain Head	DWORD	n/a	22h	1Ah
# Protected FCBs	WORD	n/a	26h	1Eh
# Block Devices	BYTE	n/a	10h	20h
NUL Device Header Block	12h	17h	28h	22h
Notes				
Offsets listed are relative to pointer returned in ES:BX by DOS int 21h/52h.				
MCB = Memory Control Block ("Arena Header").				
DCB = Device Control Block or System File Table.				
BPS = Bytes per Sector.				
CDS = Current Directory Structure.				
FCB = File Control Block.				
(1) = LASTDRIVE under v3+.				
(2) = Points to disk buffer information under DOS 4+.				

Table 27.1 *Structure of configuration variable table (CVT — "List Of Lists").*

Installable device drivers, however, must be specified in `config.sys`. Whatever their type, DOS places all drivers in a one-way linked list (chain) of device header blocks (see Listing 27.1). The first member of the list is always the NUL device, a sort of dummy device driver. To access character devices, DOS traverses the driver chain starting with the NUL device. This behavior means that you can supersede any default character driver by specifying a driver with the same name in `config.sys`. Your driver will appear before any of the default drivers in the chain, so DOS will encounter it first.

DOS locates the driver chain immediately after a data area called the Configuration Variable Table (CVT), commonly known as the "List Of Lists" (LOL) (see Table 27.1). You can access the CVT by calling DOS function `52h`, which returns a pointer to the CVT in `ES:BX`. The NUL

Request block on input			
Offset	**Size**	**Type**	**Description**
02h	BYTE	Chr/Bl	Command code (0 = init)
12h	DWORD	Chr/Bl	Pointer to command line (1st byte after "=" in CONFIG.SYS line)
16h	BYTE	Block	1st available drive number (DOS 3+)
17h	BYTE	Chr/Bl	1 = Enable CONFIG.SYS error message 0 = Disable CONFIG.SYS error message (4+)
Request block on output			
Offset	**Size**	**Type**	**Description**
03h	WORD	Chr/Bl	Status (OK = 0, Error = 8000h + code)
0Dh	BYTE	Block	Number of units to be controlled by this driver
0Eh	DWORD	Chr/Bl	Break address (End of driver image)
12h	DWORD	Block	Pointer to array of pointer (offset only) to BPB table(s). Array size = number of untis controlled.

Table 27.2 ***Items passed in request block for driver command code 0 (initialize driver).***

device header is located immediately following (or in later versions, inside) the CVT. The makeup of this undocumented structure has changed to some extent with each release of DOS. While you can easily account for this variation in present code, the usual warnings apply regarding the use of undocumented features; namely, that these features are not guaranteed to be present in later DOS versions.

After loading a driver specified in *config.sys*, DOS calls command code routine 0 to initialize the driver before linking it into the driver chain. If it encounters problems during driver initialization, DOS abandons the attempt to load the current driver and proceeds to the next driver, if any, in *config.sys*. Table 27.2 summarizes the data items DOS passes to the initialization routine (command code 0), as well as the items expected on return. Listing 27.1 outlines the structure of the request block, which is used to pass data to and from the driver command-code routines.

As Table 27.2 implies, loading a character device driver involves less labor than loading a block-type driver. Except for the status code, DOS expects only a break address on return from initializing a character driver. The break address tells DOS how much of the driver to keep in memory, and also where to load the next driver. DOS links a device driver into the driver chain by assigning the pointer field of the previous driver to point to the header of the new driver. If there are no other drivers to load, the pointer field of the header must contain *-1L* (*0ffffffffh*), which indicates the end of the chain. Once linked into the driver chain, you can access a character driver by calling DOS *int 21h/3dh* (DOS Open File). Using the returned handle, you can pass data to and from the device with DOS *Read* and *Write* functions (*3fh* and *40h*, respectively), just as you would with a disk file. DOS accesses block-type drivers differently, since these drivers have no name in the header block.

Loading and initializing a block driver involves more complex considerations. In addition to the command code and command-line address (the command line after the "=" in *config.sys*), DOS also passes the first free drive number to the driver's initialization routine (see Table 27.2). This lets the block driver know what drive letter(s) DOS will assign to it. DOS appears to acquire this number by searching an array of structures, each element of which is a Current Directory Structure (CDS) (see Table 27.3). Among other items, the CDS contains a *drive:path* string that specifies the current directory for a disk. The CDS array contains *LASTDRIVE* elements (default = 5), a quantity you can specify in `config.sys`. The CDS table also accounts for `SUBST`ed and network-redirected drives. DOS uses the index of the first free entry in the CDS table as the first available drive number, and passes it to the driver's initialization routine. For example, if the first free drive is three (`D:`) and the driver will control two units (devices), then the drive letters assigned to the drive will be `D:` and `E:`. Note that the CDS table does not exist under DOS v2.x.

DOS requires that a block driver's initialization routine return, in addition to the break address, a pointer to an array of pointers to BIOS Parameter Blocks (BPB), as well as the size of the pointer array (see Table 27.2). Since a single block driver can control more than one block device, the array of BPB pointers must contain one pointer for each device to be controlled. However, there need not be a different BPB for each device; each element of the array may point to the same BPB if

		Offsets	
Field	**Size**	**v3**	**v4.5**
Drive designator and ":"	WORD	00h	00h
CWD (path) string	41h	02h	02h
Drive status bitmap	WORD	43h	43h
Pointer to DPB	DWORD	45h	45h
CWD 1st cluster	WORD	49h	49h
Unknown	DWORD	4Bh	4Bh
Slash offset in CWD	WORD	4Fh	4Fh
Unknown	BYTE	n/a	51h
Ptr IFS driver	DWORD	n/a	52h
Unknown	WORD	n/a	56h
Notes			
CDS does not exist under DOS v2.			
CWD = Current Working Directory			
DPB = Disk (or Drive) Parameter Block			
IFS = Installable File System			

Table 27.3 *Current Directory Structure (CDS).*

needed. Table 27.4 outlines the structure of a BPB, which contains the physical characteristics of the block device(s) to be controlled.

If the block driver was successfully initialized, DOS proceeds to update the CDS table and also builds a Drive Parameter Block (DPB) (see Table 27.5) at the break address. DOS executes this procedure once for each device to be controlled by the driver. Besides duplicating information obtained from the BPB, the DPB contains a pointer to the header of the corresponding block driver. DOS also places all DPBs in a one-way linked list. A pointer to the head of the DPB list is contained in the CVT. You can locate the DPB for a specified drive by calling `int 21h/32h` with `DL` containing the drive number (default = 0, A = 1, etc.).

		Offsets		
Field	**Size**	**v2**	**v3**	**v4.5**
# Bytes per sector	WORD	00h	00h	00h
# Sectors per cluster	BYTE	01h	01h	02h
# Reserved sectors	WORD	03h	03h	03h
# FATs	BYTE	05h	05h	05h
Root directory size	WORD	06h	06h	06h
Total media sectors (1)	WORD	08h	08h	08h
Media descriptor	BYTE	0Ah	0Ah	0Ah
# Sectors per FAT	WORD	0Bh	0Bh	0Bh
# Sectors per track	WORD	n/a	0Dh	0Dh
# Heads	WORD	n/a	0Fh	0Fh
# Hidden sectors	DWORD	n/a	11h	11h
Reserved area	0Bh	n/a	15h	n/a
Total media sectors (2)	DWORD	n/a	n/a	15h
Reserved area	06h	n/a	n/a	19h
Notes				
FAT = File Allocation Table				
If (1) = 0, then (2) has total sector count				

Table 27.4 ***Structure of BIOS parameter block and boot record extensions.***

The function returns a pointer to the drive's DPB table in `DS:BX`. To build the DPB, DOS must translate and update information from the BPB into run-time DPB format. You have access to this same procedure by calling

		Offsets		
Field	**Size**	**v2**	**v3**	**v4+**
Drive number (A = 0)	BYTE	00h	00h	00h
Device driver unit #	BYTE	01h	01h	01h
Bytes/sector	WORD	02h	02h	02h
(Sectors/Cluster) - 1	BYTE	04h	04h	04h
Cluster shift (1)	BYTE	05h	05h	05h
1st FAT sector	WORD	06h	06h	06h
Copies of FAT	BYTE	08h	08h	08h
Root directory entries	WORD	09h	09h	09h
1st data sector	WORD	0Bh	0Bh	0Bh
Max cluster + 1	WORD	0Dh	0Dh	0Dh
Sectors/FAT	BYTE	0Fh	0Fh	n/a
Sectors/FAT	WORD	n/a	n/a	0Fh
1st sector root dir	WORD	10h	10h	11h
Device driver header	DWORD	12h	12h	13h
Media descriptor	BYTE	16h	16h	17h
Drive access flag	BYTE	17h	17h	18h
Pointer to next DPB	DWORD	18h	18h	19h
CWD 1st cluster #	WORD	1Ch	n/a	n/a
CWD path (ASCIIZ)	40h	1Eh	n/a	n/a
Last allocated cluster	WORD	n/a	1Ch	1Dh
Number free clusters	WORD	n/a	1Eh	1Fh
Notes				
FAT = File Allocation Table				
CWD = Current Working Directory Pointer to DPB returned in DS:BX by DOS int 21h/31h, where DL = drive number (default = 0, A = 1).				
(1) = $\log_2$ (sectors/cluster)				

Table 27.5 ***Structure of Drive Parameter Block.***

`int 21h/53h`. To use this function, load `DS:SI` with the address of a BPB table (returned by the driver in the request block) and point `ES:BP` to the area that will contain the new DPB. The function does not appear to return an indication of error in the carry flag.

Designing a Self-Loading Device Driver

To construct a driver with self-loading capabilities, you have to imitate what DOS does within the context of a normal `.com` or `.exe` format file. The result is a hybrid program that combines features of device drivers and Terminate-and-Stay-Resident (TSR) programs, giving you the best of both worlds. On the one hand, a TSR-like device driver offers many advantages in flexibility (which I describe fully later). On the other, a driver-like TSR that employs some of the well-documented features of DOS is easier to live with. Coding a TSR this way helps to avoid some of the problems that arise from lack of standardization of TSRs.

You can think of the self-loading code as a shell that surrounds the resident driver code. The shell exists only to initialize the driver and place it in the proper perspective so that DOS knows where to look for it. Once finished with its work, the shell drops away. Listing 27.2 (`sldd.asm`) presents a self-loading driver shell that implements this strategy. Because this module contains the specialized startup code that performs the required initialization tasks, it must appear first in the link order. Listing 27.2 also contains the device header and entry points for the strategy and interrupt routines.

The resident portion of the driver follows standard Microsoft/Borland conventions for naming and ordering segments. The resident driver code thus exists within the framework of a normal `.exe` format program. This means that you can use C or another suitable high-level language to write the unique parts of the driver, then you simply link it with the boilerplate code presented here. Listing 27.1 (`sldd.inc`) contains the segment definitions used by the driver.

The segments defined in Listing 27.1 include a special segment named `_INIT`, which falls after all other segments. This segment contains code that initializes the resident portion of the driver code in a manner similar to many TSRs. The `_INIT` segment is designed to jettison itself once setup has been completed. Although you load the driver as an `.exe` program, the code within `_INIT` acts as a separate `.com`-format program tacked on to the end of the main resident program. All data and code occur in a single segment, so both `CS` and `DS` are normally `ASSUME`d relative to the `_INIT` segment throughout the startup procedure. The only exception occurs when the startup code must refer to data in the resident segments.

Figure 27.1 displays a memory map of the driver. Since a self-loading device driver has to be loaded as either a `.com` or (in this case) an `.exe`

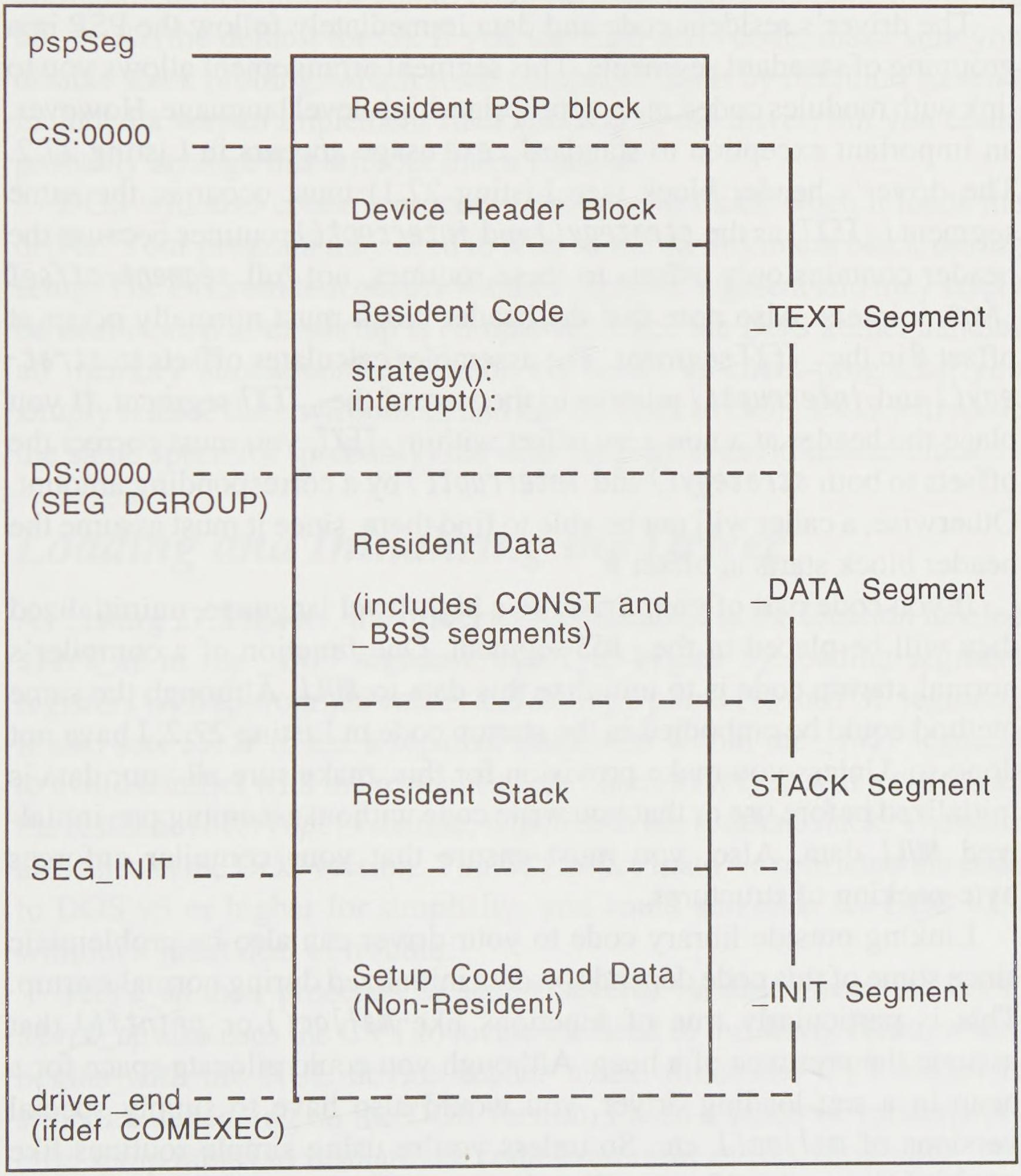

pspSeg
Resident PSP block
CS:0000
Device Header Block
Resident Code
strategy():
interrupt():
_TEXT Segment
DS:0000
(SEG DGROUP)
Resident Data
(includes CONST and
_BSS segments)
_DATA Segment
Resident Stack
STACK Segment
SEG_INIT
Setup Code and Data
(Non-Resident)
_INIT Segment
driver_end
(ifdef COMEXEC)

Figure 27.1 *Memory map of driver.*

file, the installed driver will be prefaced by a Program Segment Prefix (PSP) block, 256 bytes long. It should be possible to re-use most of this block as buffer space or for data storage at runtime, but I am unsure about the correctness of treating the entire PSP in this manner. At a minimum, you should be able to reuse the 128-byte command tail at `PSP:0080h` without reservation once setup has been completed. If your resident code uses open files, however, you will need to preserve the area beginning at `PSP:0018h`, which contains the file handle table and other sensitive items.

The driver's resident code and data immediately follow the PSP in a grouping of standard segments. This segment arrangement allows you to link with modules coded in an appropriate high-level language. However, an important exception to standard `.exe` usage appears in Listing 27.2. The driver's header block (see Listing 27.1) must occur in the same segment (`_TEXT`) as the `strategy()` and `interrupt()` routines because the header contains only offsets to these routines, not full `segment:offset` (`far`) pointers. Also note that the header block must normally occur at offset `0` in the `_TEXT` segment. The assembler calculates offsets to `strategy()` and `interrupt()` relative to the start of the `_TEXT` segment. If you place the header at a non-zero offset within `_TEXT`, you must correct the offsets to both `strategy()` and `interrupt()` by a corresponding amount. Otherwise, a caller will not be able to find them, since it must assume the header block starts at offset `0`.

If you code part of your driver in a high-level language, uninitialized data will be placed in the `_BSS` segment. One function of a compiler's normal startup code is to initialize this data to `NULL`. Although the same method could be embodied in the startup code in Listing 27.2, I have not done so. Unless you make provision for this, make sure all your data is initialized before use or that you write code without assuming pre-initialized `NULL` data. Also, you must ensure that your compiler enforces byte-packing of structures.

Linking outside library code to your driver can also be problematic since some of this code depends on data initialized during normal startup. This is particularly true of functions like `malloc()` or `printf()` that assume the presence of a heap. Although you could allocate space for a heap in a self-loading driver, you would also have to supply special versions of `malloc()`, etc. So unless you're using simple routines like `strlen()`, you should generally plan on coding your own versions of standard library routines.

Although not strictly necessary, a resident `STACK` segment makes life easier, especially if you're working with a stack-intensive, high-level language such as C. Running your device driver from the caller's stack (usually the DOS kernel) places unnecessary restrictions on your code (the DOS stack is reportedly as short as 50 to 60 bytes). A resident stack contributes to your peace of mind and lets you work with automatic, stack-based variables instead of using global data exclusively. This arrangement also allows you to compile your high-level code assuming `SS == DS` (the default for C). If you use high-level code, make sure you disable stack probing, which some compilers insert by default. I haven't included a way to implement stack probing in the driver, but you could probably arrange this without much trouble.

DOS will also create a process environment block when it loads the driver. Your program may need to refer to the environment block during setup. The environment occurs within a separate segment and may safely be deallocated after startup is completed. Since the DOS kernel handles all memory allocation/deallocation on behalf of client programs, you simply release the environment and hand it back to DOS. DOS will reuse the same space for (probably) the next program's environment block.

Loading and Initializing the Driver

As Listing 27.2 shows, the driver starts execution at the location labeled `start_up` in the `_INIT` segment. `start_up` begins by loading segment registers with appropriate values and saving a pointer to the PSP segment. It also sets `SS:SP` to use a separate stack area within the `_INIT` segment to avoid conflict with the resident stack. Later, `start_up` will have to call the resident `interrupt()` routine, which uses the resident stack. The code also checks the DOS version. You will notice that I've restricted the code to DOS v3 or higher for simplicity; you could add code for DOS v2.x without a great deal of trouble.

`start_up` then proceeds to access several variables from the CVT. `start_up` also uses the CVT to locate the head of the driver chain, which begins with the NUL device header block. Since the CVT varies in structure depending on the DOS version, I used a series of version-specific setup tables to insulate the code from most of this variability. The setup tables contain offsets into the CVT, as well as the size of the CDS. `start_up` points register `DI` to the correct setup table based on the DOS version. `start_up` can then unpack the correct item from the setup table without further regard to the DOS version. If the structure of the CVT changes in future versions of DOS, you can easily update the setup tables.

To verify that it has found the head of the driver chain, `start_up` compares the presumed location of NUL with the known name of the NUL device. The program aborts immediately if a match is not found. The driver must also guard against repeated invocations of itself. Since the driver eventually links itself into the driver chain, you can let DOS check for the presence of an earlier invocation by issuing an `Open File/Device` call (`int 21h/3dh`). This function also provides application programs with a well-documented method of determining the load status of any driver/TSR that links itself into the driver chain. You use `int 21h/3dh` for this purpose exactly as you would when opening a disk file. Be sure to set the open mode in `AL` to read only (`AL = 0`). If the carry flag is set on return, the device has not been installed yet. If the carry flag is clear, either the device is installed or a disk file exists with the same name as your device.

`start_up` must then call the driver's `interrupt()` routine (also in Listing 27.2) to initialize the driver, exactly as DOS would do. It fills a request header block (`cmdPkt`) with appropriate values and first calls `strategy()`, which simply saves a pointer to `cmdPkt`. Immediately after, `start_up` calls `interrupt()` to perform the actual initialization. To fill the request header block, `start_up` must also call `new_drive()` to get the next available drive number. Although this item concerns only block-type drivers, `start_up` fills `cmdPkt.rh_drive` with the value returned by `new_drive()`, regardless of the type of driver to be loaded. If `interrupt()` returns an error code, `start_up` immediately aborts the program.

The driver `interrupt()` routine for the most part follows standard procedure. As noted earlier, it sets `SS:SP` to point to the resident stack. It also points `ES:DI` to the request packet and then calls `exec_command()`, found in `cmd.asm` (Listing 27.3). `exec_command()` routes the call to the correct command code routine. It uses the command code as an index into a jump table. `exec_command()` calls each command code routine with `ES:DI` pointing to the request packet. It also expects each routine to return a status code in `AX`, which will be 0 if no errors were encountered.

Standard device drivers loaded from `config.sys` are restricted to DOS `int 21h` functions `01h-0ch` and `30h` during execution of the `init()` routine. However, the `init()` routine in Listing 27.3 is not subject to these restrictions. The self-loading capability allows you to execute any DOS function inside `init()`. Moreover, instead of imitating DOS by calling `init()` via the standard `strategy()` and `interrupt()` entry points, you could place `init()` within the `_INIT` segment. This would allow `init()` to drop away with the `_INIT` segment when the driver goes resident, which may be important if you have lengthy initialization code. On the other hand, keeping `init()` in its normal place makes it easier to convert the driver back to a standard format (one loaded from `config.sys`). This might be the better choice if you're using the self-loading method for its enhanced debugging facility and intend to convert to standard format later.

The code in Listing 27.3 contains no provision for calling command code routines written in a high-level language, but you can implement this easily with a few minor modifications. First, replace the function names in the jump table (`cmdTab`) with command code routines you've defined in a high-level module. Then, to allow your routines access to the request header block, pass them a (`far`) pointer to it in `exec_command()`:

```
push es    ;request block segment
push di    ;request block offset
call word ptr [bx].cmdTab ;call routine
add sp,4    ;adjust stack (C functions)
```

Alternatively, you could assign `ES:DI` to a pointer object, one that your high-level routines could access globally. As before, each of the command code routines must return a status code (usually in `AX`).

The code in Listing 27.4 (`initblk.asm`) handles the special initialization requirements of block-type device drivers. Designed as an include file, this module is inserted at the appropriate point in `start_up` and can be left out if you're building a character-type driver. For the most part, it follows a procedure very similar to that outlined earlier for DOS. Its major task is to build a Disk Parameter Block (Table 27.5) immediately after the driver image in memory. It builds one DPB table for each unit to be controlled by the driver. It must also update the CDS array (Table 27.3) and the block device count in the CVT (Table 27.1).

When `start_up` safely completes all initializations, it inserts the new driver into the driver chain immediately after the NUL device. `start_up` must also compute the size of the resident portion of the driver on the basis of the returned break address and any modifications made to the break address in `initblk.asm` (Listing 27.4). The total resident size is expressed in terms of paragraphs (1 paragraph = 16 bytes). `start_up` then completes the installation by calling the DOS *Keep* (TSR) function, `int 21h/31h`, where `DX` contains the number of resident paragraphs. At this point, the driver is installed in memory and will function as a normal device driver, at least from DOS's perspective.

Debugging and Testing the Driver

The difficulties involved in debugging a standard device driver that DOS loads from `config.sys` are well known and undoubtedly contribute to the general perception of drivers as being "difficult." A software-only debugger used in the conventional manner usually won't work. Remote debugging via a serial link to a second machine or with hardware-assisted apparatus can be effective if you have access to such facilities. Lacking these advanced capabilities, however, most programmers resort to testing the driver code by embedding it in a transient test program, as they would a TSR. Though constructive when properly used, this technique will not uncover problems that may arise only when you operate the driver under normal conditions. Recoding your device driver along the lines presented in this article can remove some of these restrictions and thereby help your debugging efforts considerably.

Listing 27.2 incorporates code that loads a copy of the `command.com` shell on top of the driver's memory image. You activate this code at assembly time by defining the `COMEXEC` macro on the assembler's command line. With `COMEXEC` defined, the driver's `init` routine (Listing 27.3) relocates the break address to the end of the `_INIT` segment, instead of at `_INIT:0000` (see Figure 27.1). This causes all transient code inside `_INIT` to remain in memory along with the driver itself. With `command.com` loaded, the driver is now pseudo-resident, and both DOS and your applications can interact with the driver as they normally would under true residency.

Typing "EXIT" on the command line unloads the command shell and returns control to `start_up`. `start_up` then unlinks the driver from the driver chain and invokes DOS Terminate Program (`int 21h/4ch`), which returns control to DOS. Once terminated, the driver no longer occupies memory. You can thus install the driver any number of times for testing; the `EXIT` command will always wipe the slate clean.

Loading the command shell in this way also means you can load a software debugger along with the driver so that it too becomes "resident." Invoke the debugger as you normally would to debug a program, supplying the driver's `.exe` filename as the debugger's argument. Once inside the debugger (at the label `start_up`), you can set breakpoints at any desired location, usually within selected command code routines. Next, invoke the debugger's `RUN` command, which will execute the remainder of the startup code and load the command shell without further intervention. You'll then find yourself at the DOS command line.

Once resident, the debugger will pop up whenever a breakpoint within the driver is triggered. Simply invoke the debugger's `RUN` command when you've finished stepping through the code. There is one important caveat, however: you must avoid situations in which breakpoints are activated as a result of calls made by the DOS kernel. This problem arises, for example, when you use DOS commands such as `TYPE` or `COPY` to test how your driver responds to a `device_write` call (command code 8). Using `fwrite()` to pass information to the driver also goes through the DOS kernel. Such cases will activate the breakpoints and pop up the debugger as always, but your system will eventually lock up, the reason being that almost all software-only debuggers make DOS calls when activated. Since the DOS kernel has activated the debugger in this case, any subsequent `int 21h` calls will reenter DOS. Unfortunately, DOS wasn't designed to allow reentrant calls, which corrupt the stack used by the DOS kernel.

The solution is to use a test bench program that imitates the DOS kernel's interaction with device drivers. Listing 27.5 (`tdrv.c`) presents a working skeleton of such a program. At present, `tdrv` simply passes a nonsense string to the driver using command code 8. Function `device_write()` in the driver (Listing 27.3) responds by writing the string to the screen using BIOS video services. If you load the debugger along with the driver as described earlier, invoking `tdrv` will trigger any breakpoints you had previously set in the driver code. `tdrv` allows you to pop up the debugger without interference from reentrant DOS calls.

`tdrv` traverses the driver chain to locate the device header for your driver. It then calls the driver's `strategy()` and `interrupt()` routines, which it accesses through the device header. The example in `tdrv` lets you test only command code 8 (`device_write()` in Listing 27.3). To test a real-world driver, you would have to modify `tdrv` extensively, adding many more test cases — which might lead you to make `tdrv` interactive. Another possibility would be to configure a request header block by parsing test data you place in a normal text file. Such a script-driven version of `tdrv` could accommodate a large number of test cases without recompiling.

A few warnings are in order when using a driver test bed of this nature. A program like `tdrv` attempts to duplicate the DOS kernel's method of making calls to the driver. You must therefore know exactly what data DOS places in the request block prior to making the call. If you debug based only on your assumptions in this regard, you will merely confirm your own ignorance. This is particularly true with respect to block-type drivers, which involve more complex connections within the kernel-to-driver interface.

To deal with this problem, you might consider equipping the driver with some simple instrumentation functions, such as data conversion and console output routines. Load your driver (with `COMEXEC` enabled) and invoke a program that will output data to the driver via the DOS kernel. At each call to `exec_command()`, call the instrumentation functions to convert and display the entire contents of the request block as filled in by DOS. As noted earlier, you cannot use a software-only debugger in this situation, and you must make sure the instrumentation code makes only BIOS or lower-level calls. To save the instrumentation output, consider sending it to a printer instead of the console. Spooling to a disk file would require complex coding to circumvent DOS reentrant limitations.

Conclusions

This article has presented one method of writing a self-loading DOS device driver. By combining traits of both device drivers and TSRs in one program, this method increases your flexibility and provides a relative degree of independence from the constraints imposed when DOS loads a driver from `config.sys`. The freedom gained also permits the use of debugging techniques comparable to those you might use when developing transient programs.

References

Dettmann, Terry, and Marcus Johnson. *DOS Programmer's Reference*. 3rd Ed. Carmel, IN: Que Corp. A good companion work to the officially sponsored *Advanced MS-DOS Programming* by Ray Duncan. Includes a good amount of reference material on undocumented areas of DOS. The reported structure of the CVT, however, conflicts with that given in Schulman's *Undocumented DOS*.

Microsoft Corp. *MS-DOS Programmer's Reference*. Redmond, WA: Microsoft Press. Provides official documentation for some previously undocumented data structures and function calls, including function `32h` and the `DPB`. The CVT, CDS and functions `52h` and `53h`, however, still remain officially untouchable.

Schulman, Andrew, et al. *Undocumented DOS*. Reading, MA: Addison-Wesley. A goldmine of information about hidden aspects of DOS and proper techniques for use thereof. Loaded with real-world example code.

Listing 27.1 sldd.inc

```
;-----------------------------------------------------------------------------------
;   Structure definitions and segment setup for self-loading MS-DOS device drivers.

;Driver command packet .....

RH_HDR  struc
rh_len      db  ?
rh_unit     db  ?
rh_cmd      db  ?
rh_status   dw  ?
rh_resrvd   db  8 dup(?)
rh_nunits   db  ?       ;#block units to be supported
rh_brkofs   dw  ?       ;load address for next
rh_brkseg   dw  ?       ;... driver
rh_bufofs   dw  ?       ;address of table of offsets
rh_bufseg   dw  ?       ;...to BPB tables
rh_drive    db  ?       ;1st free drive# (A=0, B=1)
RH_HDR  ends

;Device driver header block .....
d_header    struc
d_next      dd      ?
d_attr      dw      ?
d_strat     dw      ?
d_intr      dw      ?
d_name      db      8 dup(?)
d_header    ends

; Specialized driver segment setup ......
NOWARN  RES     ;disable reserved-word warnings for "CONST" and "STACK"

_TEXT   segment byte    public  'CODE'
_TEXT   ends
_DATA   segment word    public  'DATA'
_DATA   ends
CONST   segment word    public  'CONST'
CONST   ends
_BSS    segment word    public  'BSS'
_BSS    ends
STACK   segment para    stack   'STACK'
STACK   ends
_INIT   segment word    public  'INIT'
_INIT   ends

WARN            ;re-enable all warnings

; Establish SMALL model segment group ....

DGROUP  group   _DATA,CONST,_BSS,STACK

wptr    equ     <word ptr>
bptr    equ     <byte ptr>
dptr    equ     <dword ptr>
```

Listing 27.2 sldd.asm

```
;--------------------------------------------------------------------------
;   Summary:    Self-loading device driver
;   Assembler:  TASM 2.0
;   Author:     T.W. Nelson
;   Version:    1.00
;   Date:       29-Jan-1992
;   Notes:
;
;   This module is "start-up" code and must appear first in the link order.
;
;   Source code Copyright (c) 1992 T.W. Nelson. May be used
;   only with appropriate acknowledgment of copyright.
;--------------------------------------------------------------------------

INCLUDE sldd.inc
ASSUME  cs:_TEXT,ds:DGROUP,es:NOTHING,ss:NOTHING
STACKSIZ    equ     256     ;driver stack size, words

_DATA   segment
Rhdr        dd  ?               ;-> request header
Caller_ss   dw  ?               ;caller's stack vitals
Caller_sp   dw  ?
__psp       dw  ?               ;program seg prefix
public __psp
_DATA   ends

STACK   segment
            dw  STACKSIZ dup(-1) ;interrupt stack
stack_top   dw  ?

public stack_top
STACK   ends

_TEXT   segment

;-------------------------------------------------------
; Resident driver header block.  IMPORTANT: Header block
; must start at offset 0 in the _TEXT segment, otherwise
; offsets in 'drv_strat' and 'drv_intr' must be corrected
; by subtracting (offset _TEXT:drv_header) from them.
;
drv_header  label byte
drv_next    dd  -1              ;-> ground (NULL)
drv_attr    dw  8000h           ;driver attributes
drv_strat   dw  _strategy       ;offset strategy()
drv_intr    dw  _interrupt      ;offset interrupt()
drv_name    db  "TEMPLATE",0    ;device name, 8 bytes

public drv_header, drv_next, drv_attr, drv_strat
public drv_intr, drv_name
extrn   exec_command:proc
```

Listing 27.2 *continued*

```
;-------------------------------------------------------
public _strategy

_strategy          proc    far
        push    ax
        push    ds
        mov     ax,seg DGROUP
        mov     ds,ax
        mov     wptr [Rhdr+2],es
        mov     wptr [Rhdr],bx
        pop     ds
        pop     ax
        ret
_strategy   endp

;-------------------------------------------------------
public _interrupt

_interrupt         proc    far
        push    ax          ;Save caller's CPU state
        push    bx          ;and stack context ....
        push    cx
        push    dx
        push    di
        push    si
        push    bp
        push    ds
        push    es
        pushf
        mov     ax,seg DGROUP         ;local data seg
        mov     ds,ax
        mov     Caller_ss,ss
        mov     Caller_sp,sp
        cli
        mov     ss,ax
        mov     sp,offset DGROUP:stack_top
        sti
        mov     bp,sp                 ;set C stack frame
        les     di,[Rhdr]             ;request header
        call    exec_command
        cli                           ;restore CPU state
        mov     ss,Caller_ss          ;and caller's stack
        mov     sp,Caller_sp
        sti
```

Listing 27.2 *continued*

```
        popf
        pop     es
        pop     ds
        pop     bp
        pop     si
        pop     di
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret
_interrupt  endp

_TEXT   ends

;----------------------------------------------------------------
; Driver load and initialization code.  The entire _INIT segment
; is jettisoned after initialization if COMEXEC is not defined.
;
_INIT   segment
ASSUME  cs:_INIT,ds:_INIT,es:NOTHING,ss:NOTHING

dosMajor    dw  0       ;DOS version #s
dosMinor    dw  0       ;
DosDataSeg  dw  0       ;DOS data segment
ofsLoL      dw  0       ;offset to DOS LoL
pBlkDevCnt  dd  0       ;-> Block device count in LoL
LastDrive   dw  0       ;LASTDRIVE value in LoL
ofsNULdev   dw  0       ;offset of NUL device header
pCDS        dd  0       ;-> Current Directory Structure
sizeofCDS   dw  0       ;sizeof a CDS[] element
pspSeg      dw  0       ;PSP segment
NULname     db  "NUL     ",0 ;NUL device name
cmdPkt      RH_HDR  <>  ;driver command packet
            dw  STACKSIZ dup(-1)  ;setup stack
setup_stack dw  ?

;Version-specific offsets for DOS data (CVT, etc.) ...
DOS30   db      10h,1Bh,17h,28h,51h
DOS31   db      20h,21h,16h,22h,51h
DOS4    db      20h,21h,16h,22h,58h     ;DOS 4+
```

Listing 27.2 *continued*

```
;byte offsets into initializer tables .....
E_pBlkDevCnt  equ   0
E_LastDrive   equ   1
E_pCDS        equ   2
E_ofsNULdev   equ   3
E_sizeofCDS   equ   4

dos_ver     db  "You need at least DOS ver 3",10,13,0
bad_dos     db  "Unknown DOS version",10,13,0
no_nul      db  "Can't find NUL device header",10,13,0
is_loaded   db  " device is already loaded",10,13,0
is_file     db  " device is a disk file",10,13,0
bad_init    db  "Driver initialization failed",10,13,0
unloaded    db  "Driver template unloaded",10,13,0
CDSfull     db  "Out of CDS entries",10,13,0
no_block    db  "Can't install block device",10,13,0

; Current Directory Structure .......
CDS struc
path    db  43h dup(?)  ;current path
flags   dw  ?           ;= 0 if entry available
pDPB    dd  ?           ;-> disk param block
sclust  dw  ?           ;starting cluster
ffff    dd  ?           ;always -1 (?)
slash   dw  ?           ;ofs '\' in current path
qmark1  db  ?           ;unknown byte  (DOS 4+)
ifs     dd  ?           ;-> ?          (DOS 4+)
qmark2  dw  ?           ;unknown byte  (DOS 4+)
CDS ends

IFDEF   COMEXEC     ;load command shell to simulate residency
comSpec         db      "c:\command.com",0
exec_params     equ     $   ;parameter block for
environ_seg     dw      ?   ;DOS EXEC function ....
command_ofs     dw      ?
command_seg     dw      ?
fcb1_ofs        dw      ?
fcb1_seg        dw      ?
fcb2_ofs        dw      ?
fcb2_seg        dw      ?
save_sp         dw      ?

ENDIF   ;COMEXEC
```

Listing 27.2 *continued*

```
;-------------------------------------------------------
; entry:    al == exit code
; exit:     n/a
;
exit2dos:
    mov     ah,4ch          ;DOS exit
    int     21h

;-------------------------------------------------------
; entry:    ds:bx - asciiz message string
; exit:     n/a
;
err_exit:
    call    putstr
    mov     al,3
    jmp     exit2dos

;-------------------------------------------------------
; entry:    ds:bx - asciiz message string
; exit:     n/a
;
putstr  proc    near
    push    ax
    push    bx
putstr1:
    mov     al,[bx]
    or      al,al           ;finished?
    jz      _px             ;yes
    mov     ah,0eh          ;BIOS write tty
    push    bx              ;save it
    xor     bx,bx           ;page 0
    int     10h
    pop     bx              ;restore bx
    inc     bx              ;increment pointer
    jmp     putstr1         ;next byte
_px:
    pop     bx
    pop     ax
    ret
putstr  endp
```

Listing 27.2 *continued*

```
;------------------------------------------------------
; entry:    n/a
; exit:     ax = index of 1st avail CDS entry, aborts
;           program if no free entries exist
; note:     pCDS, sizeofCDS must be known before call
;
new_drive   proc    near
        push    es
        push    bx
        les     bx,pCDS         ;point to table head
        mov     ax,0            ;index of 1st CDS entry
n1:     cmp     es:[bx].flags,0 ;entry available?
        je      nex             ;yes, quit
        add     bx,sizeofCDS    ;else, -> next entry
        inc     ax              ;next index
n5:     cmp     ax,LastDrive    ;end of loop?
        jl      n1              ;no, continue
        mov     bx,offset CDSfull ;yes, no entries free
        jmp     err_exit          ;abort the program
nex:    pop     bx
        pop     es
        ret
new_drive   endp

;------------------------------------------------------
start_up:

    mov     ax,cs               ;Recover segment
    mov     ds,ax               ;registers and PSP seg
    mov     pspSeg,es
    cli
    mov     ss,ax
    mov     sp,offset setup_stack
    sti
    mov     ax,seg DGROUP
    mov     ds,ax
    ASSUME  ds:DGROUP           ;get -> PSP for
    mov     __psp,es            ;resident portion
    push    cs
    pop     ds
    ASSUME  ds:_INIT
```

Listing 27.2 *continued*

```
; ----- Check and save DOS version -------------------
    mov     ax,3000h            ;DOS get version
    int     21h
    cmp     al,3                ;at least ver 3?
    jge     saveDOSver          ;yes, bypass
    mov     bx,offset dos_ver   ;no ....
    jmp     err_exit            ;quit
saveDOSver:
    xor     bx,bx
    mov     bl,al
    mov     dosMajor,bx
    mov     bl,ah
    mov     dosMinor,bx

; ----- Grab data from CVT (List of Lists) ------------
    mov     ah,52h
    int     21h                 ;es:bx -> LoL
    cmp     dosMajor,3
    ja      dos_4x
    cmp     dosMinor,0
    ja      dos_31
    mov     di,offset DOS30
    jmp     short go_CVT
dos_31:
    mov     di,offset DOS31
    jmp     short go_CVT
dos_4x:
    mov     di,offset DOS4
    cmp     dosMajor,6
    jbe     go_CVT
    mov     bx,offset bad_dos
    jmp     err_exit
go_CVT:
    mov     DosDataSeg,es
    mov     ofsLoL,bx
    xor     ax,ax
    mov     al,[di].E_pBlkDevCnt
    add     ax,bx
    mov     wptr pBlkDevCnt[2],es
    mov     wptr pBlkDevCnt[0],ax
    xor     ax,ax
    mov     al,[di].E_LastDrive
    mov     si,ax
    mov     al,bptr es:[bx+si]
    mov     LastDrive,ax
    xor     ax,ax
```

Listing 27.2 *continued*

```
    mov     al,[di].E_pCDS
    mov     si,ax
    mov     ax,wptr es:[bx+si]         ;-> ofsCDS
    mov     wptr pCDS[0],ax
    mov     ax,wptr es:[bx+si+2]       ;-> segCDS
    mov     wptr pCDS[2],ax
    xor     ax,ax
    mov     al,[di].E_ofsNULdev
    add     ax,bx
    mov     ofsNULdev,ax
    xor     ax,ax
    mov     al,[di].E_sizeofCDS
    mov     sizeofCDS,ax

; ----- Verify that NUL device is present ------------
    mov     si,offset NULname   ;ds:si
    mov     di,ofsNULdev        ;es:di -> device hdr
    add     di,d_name           ;-> device name
    mov     cx,8                ;compare count
    repz    cmpsb               ;get 1st non-match
    jz      chk_load            ;NUL device present
    mov     bx,offset no_nul    ;can't find NUL
    jmp     err_exit

; ----- Check driver load status ---------------------
chk_load:
    mov     ax,seg _TEXT
    mov     ds,ax
    mov     dx,offset _TEXT:drv_name ;ds:dx = our name
    mov     ax,3d00h            ;DOS Open File, rdonly
    int     21h
    jc      init_driver         ;OK, driver not loaded
    mov     bx,dx               ;not OK,
    call    putstr              ;print device name
    push    cs
    pop     ds                  ;fix ds
    mov     bx,ax               ;bx = open handle
    mov     ax,4400h            ;DOS IOCTL,
    int     21h                 ;get device info
    test    dx,0080h            ;is it a file?
    jnz     already_loaded      ;no, driver installed
    mov     bx,offset is_file   ;yes, file w/ same name
    jmp     err_exit            ;quit
already_loaded:
    mov     bx,offset is_loaded ;and loaded msg
    jmp     err_exit            ;quit
```

Listing 27.2 *continued*

```
; ----- Build cmd packet and initialize driver --------
init_driver:
    push    cs
    pop     ds                  ;fix ds
    mov     cmdPkt.rh_cmd,0     ;driver INIT command
    mov     cmdPkt.rh_len,size RH_HDR
    mov     cmdPkt.rh_unit,0
    mov     es,pspSeg
    mov     di,81h              ;es:di -> PSP cmd line
d1: cmp     bptr es:[di],' '
    jne     d2                  ;skip leading spaces
    inc     di
    jmp     d1
d2: mov     cmdPkt.rh_bufofs,di
    mov     cmdPkt.rh_bufseg,es
    call    new_drive           ;don't know yet if
    mov     cmdPkt.rh_drive,al  ;it's a block device
    push    cs
    pop     es
    mov     bx,offset cmdPkt        ;es:bx -> cmd pkt
    call    far ptr _strategy
    call    far ptr _interrupt
    test    cmdPkt.rh_status,8000h  ;error?
    jz      init_block              ;no, continue
    mov     bx,offset bad_init      ;yes, bail out
    jmp     err_exit
init_block:
    include initblk.asm         ;init block device

; ----- Patch driver into existing chain --------------
    push    ds
    mov     es,DosDataSeg
    mov     bx,ofsNULdev
    mov     ax,seg _TEXT
    mov     ds,ax
    ASSUME  ds:_TEXT
    mov     dx,wptr es:[bx+2]   ;dx:ax == next driver
    mov     ax,wptr es:[bx+0]   ;after NUL device.
    mov     wptr drv_next[0],ax ;make ours point to
    mov     wptr drv_next[2],dx ;what NUL pointed to.
    mov     wptr es:[bx+0],offset _TEXT:drv_header
    mov     wptr es:[bx+2],ds   ;point NUL to us
    pop     ds
    ASSUME  ds:_INIT

IFNDEF  COMEXEC                 ;release environment
    mov     es,pspSeg           ;get environ seg
    mov     es,es:[2ch]         ;in PSP block
    mov     ah,49h              ;DOS free memory
    int     21h
ENDIF
```

Listing 27.2 *continued*

```
; ----- Compute resident program size -----------------
    mov     dx,cmdPkt.rh_brkofs
    mov     cx,4
    shr     dx,cl
    inc     dx
    add     dx,cmdPkt.rh_brkseg
    sub     dx,pspSeg           ;#resident paragraphs

IFDEF   COMEXEC
; ----- Load Command.com to simulate residency --------
    mov     es,pspSeg           ;beg segment address
    mov     bx,dx               ;total resident size
    mov     ax,4a00h            ;DOS SETBLOCK
    int     21h                 ;return unused memory
    mov     di,0080h            ;es:di -> command line
    mov     bptr es:[di],0      ;cmd length = zero
    mov     dx,offset comSpec
    mov     cx,es:[2ch]         ;grab environment seg
    mov     environ_seg,cx
    mov     command_ofs,di
    mov     command_seg,es
    mov     fcb1_ofs,5ch
    mov     fcb1_seg,es
    mov     fcb2_ofs,6ch
    mov     fcb2_seg,es
    mov     ax,4b00h            ;DOS EXEC program
    mov     save_sp,sp
    push    cs
    pop     es
    mov     bx,offset exec_params
    int     21h                 ;load COMMAND.COM
    mov     ax,cs               ;restore regs
    mov     es,ax
    mov     ds,ax
    cli
    mov     ss,ax
    mov     sp,save_sp
    sti
```

Listing 27.2 *continued*

```
; ----- Remove driver from device chain ---------------
        push    ds
        mov     es,DosDataSeg          ;point to NUL device
        mov     bx,ofsNULdev
        mov     ax,seg _TEXT
        mov     ds,ax
        ASSUME  ds:_TEXT
        mov     ax,wptr drv_next[0] ;dx:ax = what
        mov     dx,wptr drv_next[2] ;we point to
        mov     wptr es:[bx+0],ax   ;make NUL device point
        mov     wptr es:[bx+2],dx   ;to it instead
        pop     ds
        ASSUME  ds:_INIT
        mov     bx,offset unloaded
        call    putstr                 ;print unloaded msg
        xor     ax,ax                  ;DOS exit code
        jmp     exit2dos               ;quit, back to DOS
driver_end label   byte              ;end of COMEXEC image
public driver_end

ELSE                                   ;true residency
        mov     ax,3100h               ;DOS TSR
        int     21h
ENDIF   ;COMEXEC

_INIT           ends
                end     start_up

;--- End of File ---------------------------------------
```

Listing 27.3 CMD.ASM

```
;---------------------------------------------------------------------
;   Summary:    Route driver interrupt call to command code routine
;   Assembler:  TASM 2.0
;   Author:     T.W. Nelson
;   Version:    1.00
;   Date:       29-Jan-1992
;   Notes:
;
;   Source code Copyright (c) 1992 T.W. Nelson. May be used
;   only with appropriate acknowledgment of copyright.
;
;---------------------------------------------------------------------

INCLUDE sldd.inc
ASSUME  cs:_TEXT,ds:DGROUP,es:NOTHING,ss:NOTHING

_DATA   segment

logon   db  "Self-loading Device Driver,",10,13
        db  "(Template) Version 1.00",10,13,10,13,'$'

;-------------------------------------------------------
; Dispatch table for command-code routines
;
cmdTab  equ     $
    dw  init            ;0 = initialize driver
    dw  media_check     ;1
    dw  build_bpb       ;2 = build BIOS param block
    dw  ioctl_read      ;3 = read io control data
    dw  device_read     ;4
    dw  nd_read         ;5 = non-destructive read
    dw  input_status    ;6
    dw  input_flush     ;7 = flush input buffers
    dw  device_write    ;8
    dw  verify_write    ;9 = write with verify
    dw  output_status   ;10
    dw  output_flush    ;11 = flush output buffers
    dw  ioctl_write     ;12 = write io control data
    dw  device_open     ;13  (DOS 3+)
    dw  device_close    ;14  (DOS 3+)
    dw  rem_media       ;15 = removable media (3+)
    dw  output_busy     ;16 = output until busy (3+)
    dw  bad_command     ;17 = not used
    dw  bad_command     ;18 = not used
    dw  generic_ioctl   ;19 = generic io control (3.2+)
    dw  bad_command     ;20 = not used
    dw  bad_command     ;21 = not used
    dw  bad_command     ;22 = not used
    dw  get_logdev      ;23 = get logical device (3.2+)
    dw  set_logdev      ;24 = set logical device (3.2+)
```

Listing 27.3 *continued*

```
;Dummy BPB for testing block driver setup .....
bpbstruc    equ     $
    dw  512             ;bytes/sector
    db  2               ;sectors/cluster
    dw  1               ;reserved sectors
    db  2               ;#fat tables
    dw  112             ;#directory entries
    dw  720             ;#logical sectors
    db  0fdh            ;media descriptor byte
    dw  2               ;#fat sectors
    dw  9               ;sectors/track
    dw  2               ;#sides
    dw  0               ;hidden sectors

bpbtab  dw  bpbstruc    ;two units, same table
        dw  bpbstruc

_DATA   ends

IFDEF   COMEXEC
_INIT   segment
extrn   driver_end:byte
_INIT   ends
ENDIF   ;COMEXEC

_TEXT   segment

;-------------------------------------------------------
; Set up call to command code routine.  On entry,
; es:di points to the request header.  Returns
; result code in request header status word.
;
public  exec_command

exec_command    proc    near
        push    es                      ;save req hdr
        push    di
        xor     bx,bx
        mov     bl,es:[di].rh_cmd       ;load cmd code
        cmp     bx,24                   ;legal?
        jle     exec1                   ;yes, continue
        call    bad_command             ;set error code
        jmp     short exec2             ;bypass call
```

Listing 27.3 *continued*

```
exec1:
        shl     bx,1                    ;cmdTab index
        call    wptr [bx].cmdTab        ;to cmd routine
exec2:
        pop     di                      ;restore req
        pop     es                      ;header
        or      ax,0100h                ;set 'done' bit
        mov     es:[di].rh_status,ax    ;assign status
        ret
exec_command    endp

;------------------------------------------------------
; Command code routines. On entry, es:di points to the
; caller's request header.  Each routine must return
; status in ax: 0 if successful or (8000h + error code)
; if function ended in error.
;
init:                   ;0 = initialize driver

; ----- return break address --------------------------
    mov     es:[di].rh_brkseg,seg _INIT
    IFDEF   COMEXEC
    mov     es:[di].rh_brkofs,offset _INIT:driver_end
    ELSE
    mov     es:[di].rh_brkofs,0
    ENDIF

; ----- return items for block driver -----------------
    mov     es:[di].rh_nunits,2
    mov     es:[di].rh_bufofs,offset _DATA:bpbtab
    mov     es:[di].rh_bufseg,seg _DATA
```

Listing 27.3 *continued*

```
; ----- display log on message -----------------------
    mov     dx,offset logon
    mov     ah,09h      ;DOS display '$' string
    int     21h
    xor     ax,ax       ;no error
    ret

media_check:         ;1
build_bpb:           ;2 = build BIOS param block
ioctl_read:          ;3 = read io control data
device_read:         ;4
nd_read:             ;5 = non-destructive read
input_status:        ;6
input_flush:         ;7 = flush input buffers
    xor     ax,ax    ;return 'no error' status
    ret
device_write:        ;8
verify_write:        ;9 = write with verify
    mov     cx,es:[di+18]     ;byte count
    les     di,es:[di+14]     ;xfer address
    jcxz    w2       ;check for runaway error
w1:
    mov     al,es:[di]
    mov     ah,0eh          ;BIOS write tty
    xor     bx,bx           ;page 0
    int     10h
    inc     di              ;increment pointer
    loop    w1              ;next byte
w2:

output_status:       ;10
output_flush:        ;11 = flush output buffers
ioctl_write:         ;12 = write io control data
device_open:         ;13  (DOS 3+)
device_close:        ;14  (DOS 3+)
rem_media:           ;15 = removable media (3+)
output_busy:         ;16 = output until busy (3+)
generic_ioctl:       ;19 = generic io control (3.2+)
get_logdev:          ;23 = get logical device (3.2+)
set_logdev:          ;24 = set logical device (3.2+)
    xor     ax,ax    ;return 'no error' status
    ret
bad_command:
    mov     ax,8003h
    ret

_TEXT   ends
        end

;--- End of file --------------------------------------
```

Listing 27.4 initblk.asm

```
;-----------------------------------------------------
;   Summary:     Setup data structures for block device
;                Include file for sldd.asm module
;   Assembler:   TASM 2.0
;   Author:      T.W. Nelson
;   Version:     1.00
;   Date:        29-Jan-1992
;   Notes:
;
;   Source code Copyright (c) 1992 T.W. Nelson. May be used
;   only with appropriate acknowledgment of copyright.
;-----------------------------------------------------

    mov     ax,seg _TEXT
    mov     es,ax
    test    es:drv_attr,8000h   ;block device?
    jz      init0               ;yes, continue
    jmp     init_x              ;else bypass
init0:
    cmp     cmdPkt.rh_nunits,0  ;any units returned?
    jne     init1               ;yes, continue
    mov     bx,offset no_block  ;no, abort
    jmp     err_exit
init1:
    cmp     cmdPkt.rh_brkofs,0  ;paragraph aligned?
    je      init2               ;yes, bypass
    mov     bx,cmdPkt.rh_brkofs ;no, align on pgraph
    mov     cx,4
    shr     bx,cl
    inc     bx
    add     cmdPkt.rh_brkseg,bx
    mov     cmdPkt.rh_brkofs,0
init2:
    xor     cx,cx               ;starting unit#
    jmp     init10              ;test loop exit
init3:
    call    new_drive           ;get next CDS entry
    push    ax                  ;save result
    les     bx,pBlkDevCnt
    inc     bptr es:[bx]        ;1 more block device
    mov     ah,32h              ;DOS get DPB pointer
    mov     dl,al               ;drive# (A=1,B=2,etc.)
    int     21h                 ;return ds:bx -> DPB
    push    ds
    pop     es                  ;es:bx -> last DPB
    mov     ax,seg _INIT
    mov     ds,ax               ;fix ds
    add     bx,24               ;-> DPB link field
    cmp     dosMajor,3
    jle     init4
    inc     bx                  ;adjust for DOS 4+
```

Listing 27.4 *continued*

```
init4:
   push    es
   push    bx                  ;save -> DPB link
   mov     es,cmdPkt.rh_bufseg
   mov     bx,cmdPkt.rh_bufofs ;es:bx -> BPB offset
   mov     si,es:[bx]          ;unpack -> BPB
   push    es
   mov     es,cmdPkt.rh_brkseg
   mov     bp,0                ;es:bp = new DPB addr
   pop     ds                  ;ds:si -> BPB
   mov     ah,53h              ;DOS xlat BPB -> DPB
   int     21h
   mov     ax,seg _INIT
   mov     ds,ax               ;fix ds
   pop     bx
   pop     es                  ;recover -> DPB link
   mov     ax,cmdPkt.rh_brkseg
   mov     wptr es:[bx+0],0    ;link offset
   mov     wptr es:[bx+2],ax   ;link segment
   pop     ax                  ;recover new drive#
   push    ax                  ;save again
   mov     bx,sizeofCDS
   mul     bl
   les     bx,pCDS
   add     bx,ax               ;-> new CDS entry
   mov     wptr es:[bx].flags,1 shl 14
   mov     ax,cmdPkt.rh_brkseg
   mov     wptr es:[bx].pDPB[0],0
   mov     wptr es:[bx].pDPB[2],ax
   mov     es:[bx].sclust,0ffffh
   mov     wptr es:[bx].ffff[0],0ffffh
   mov     wptr es:[bx].ffff[2],0ffffh   ;-1L
   mov     es:[bx].slash,2
   cmp     dosMajor,3
   jle     init5
   mov     es:[bx].qmark1,0    ;initialize
   mov     es:[bx].qmark2,0    ;DOS 4+ entries
   mov     wptr es:[bx].ifs[0],0
   mov     wptr es:[bx].ifs[2],0
```

Listing 27.4 *continued*

```
init5:                               ;initialize new DPB
    mov     es,cmdPkt.rh_brkseg
    mov     bx,0                     ;-> new DPB
    pop     ax                       ;recover new drive#
    mov     es:[bx],al
    pop     cx                       ;recover unit#
    mov     es:[bx+1],cl
    inc     cx                       ;next unit#
    cmp     dosMajor,3
    jle     init6
    inc     bx                       ;adjust for DOS 4+
init6:
    mov     wptr es:[bx+12h],offset _TEXT:drv_header
    mov     wptr es:[bx+14h],seg _TEXT
    mov     wptr es:[bx+18h],0ffffh ;set end of DPB
    mov     wptr es:[bx+1ah],0ffffh ;chain
    add     cmdPkt.rh_brkseg,2         ;2 pgraphs for DPB
    add     cmdPkt.rh_bufofs,2         ;-> next BPB
init10:
    cmp     cmdPkt.rh_nunits,0         ;more block units?
    je      init_x                     ;no, quit
    dec     cmdPkt.rh_nunits           ;yes, continue
    push    cx                         ;save unit#
    jmp     init3
init_x:

;----- End of file ------------------------------------
```

Listing 27.5 `tdrv.c`

```
/* ---------------------------------------------------
 * Summary:        Device driver test bed
 * Author:         T.W. Nelson
 * Compile options:
 * Version:        1.00
 * Date:           29-Jan-1992
 * Notes:
 *
 * MUST compile with byte-packing enabled
 * ---------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

char   Dname[] = "TEMPLATE";  //name to search for
size_t DosDataSeg;     //DOS data segment
size_t ofsLoL;         //offset to DOS LoL
size_t ofsNULdev;      //offset to NUL device header
size_t dosMajor;       //DOS version #s
size_t dosMinor;
void (far *strat)(void);    //-> strategy routine
void (far *intrp)(void);    //-> interrupt routine
union  REGS   regs;
struct SREGS  sregs;

struct d_hdr    {
    struct d_hdr far *d_next;  //-> next header
    size_t d_attr;
    size_t d_strat;
    size_t d_intr;
    char   d_name[8];
    };
```

Listing 27.5 *continued*

```
char buf[] =
    "Bothered by unsightly bathtub rings?\r\n"
    ".... Use NEW Duzz-All Cleanser!\r\n\n";

struct reqhdr {
    char    len;
    char    unit;
    char    cmdcode;
    size_t  status;
    char    reserve[8];
    char    media;
    void    far *xfer;
    size_t  count;
    size_t  sector;
    } cmdpack = { sizeof(struct reqhdr), 0, 8, 0, "",
                0, (void far *) buf, sizeof(buf), 0 };

#define GROUND  ((struct d_hdr far *) -1L)

/* ----------------------------------------------- */
void    err_exit( char *errmsg )    {
    printf( "%s\n\n", errmsg );
    exit(1);
}

void   get_DOS_version(void)        {
    regs.x.ax = 0x3000;
    intdos( &regs, &regs );
    dosMajor = (size_t) regs.h.al;
    dosMinor = (size_t) regs.h.ah;

    if( dosMajor < 2 )
        err_exit( "Better get a DOS upgrade.");
}
```

Listing 27.5 *continued*

```
void   get_NUL_device(void)         {
    int     displacement;

    regs.x.ax = 0x5200;     //DOS get list of lists
    intdosx( &regs, &regs, &sregs );
    DosDataSeg = sregs.es;
    ofsLoL = regs.x.bx;

    switch( dosMajor )  {
        case 2: displacement = 0x17;  break;
        case 3:
                if( dosMinor == 0 )
                        displacement = 0x28;
                else
                        displacement = 0x22;
                break;
        case 4:
        case 5:
        case 6: displacement = 0x22;  break;
        default:    err_exit( "Unknown DOS version" );
    }

    ofsNULdev = ofsLoL + displacement;
}

void    find_device_header(void)    {
    struct d_hdr far *pdrv;
    size_t d_seg, d_ofs;
    register unsigned i;

    for( pdrv = MK_FP( DosDataSeg, ofsNULdev );
        pdrv->d_next != GROUND; pdrv = pdrv->d_next ) {

        for( i = 0; i < 8; i++ )    {
            if( pdrv->d_name[i] != Dname[i] )
                break;      //name doesn't match
        }

        if( i == 8 )    {   //found a match
            d_seg = FP_SEG( pdrv );
            d_ofs = FP_OFF( pdrv );
            strat = MK_FP(d_seg,d_ofs + pdrv->d_strat);
            intrp = MK_FP(d_seg,d_ofs + pdrv->d_intr);
            return;
        }
    }

    printf("Can't find %s device header", Dname);
    err_exit("");
}
```

Listing 27.5 *continued*

```
main( int argc, char **argv )      {
    regs.x.ax = 0x3d00;      //DOS OPEN file, read only
    regs.x.dx = (unsigned) Dname;
    intdos( &regs, &regs );

    if( regs.x.cflag )  {
            printf( "%s not installed.", Dname );
            err_exit("");
    }

    regs.x.bx = regs.x.ax;       //handle
    regs.x.ax = 0x4400;      //DOS get IOCTL info
    intdos( &regs, &regs );

    if( !(regs.x.dx & 0x0080) )  {
            printf( "%s is a file.", Dname );
            err_exit("");
    }

    regs.x.ax = 0x3e00;      //DOS close handle
    intdos( &regs, &regs );
    get_DOS_version();
    get_NUL_device();
    find_device_header();
    _ES = FP_SEG( (void far *) &cmdpack );
    _BX = FP_OFF( (void far *) &cmdpack );
    (*strat)();     //Call the device driver....
    (*intrp)();     //.... just like DOS does it

    return 0;
}

/* ----- End of File ------------------------------ */
```

Chapter 28

Interfacing a Windows Program to a Real-Mode Device Driver

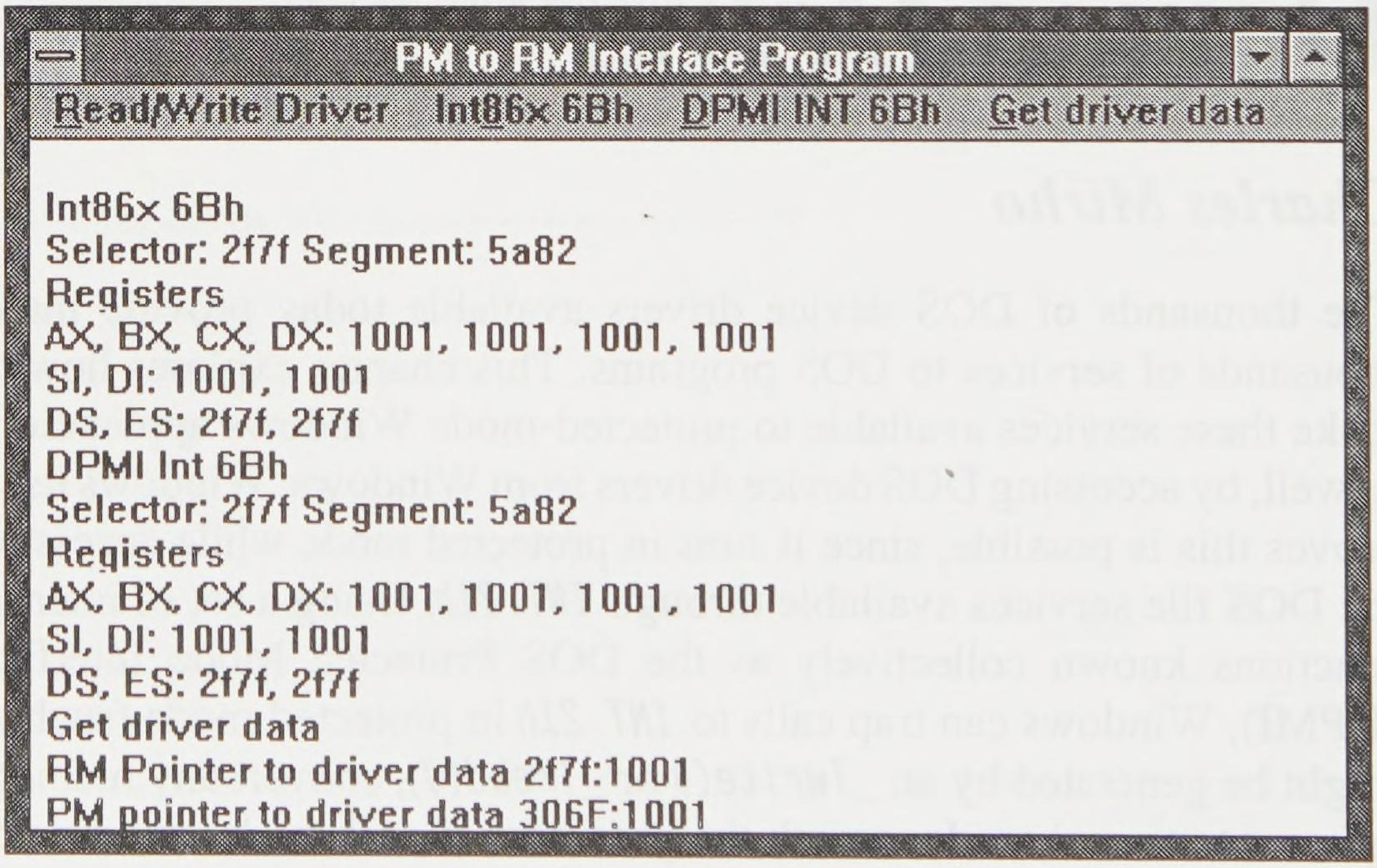
Charles Mirho

The thousands of DOS device drivers available today provide many thousands of services to DOS programs. This chapter explores how to make these services available to protected-mode Windows applications as well, by accessing DOS device drivers from Windows. Windows itself proves this is possible, since it runs in protected mode while accessing the DOS file services available through `INT 21h`. Using a set of internal functions known collectively as the DOS Protected Mode Interface (DPMI), Windows can trap calls to `INT 21h` in protected mode (such as might be generated by an `_lwrite()` or `_lread()`), convert any memory references to real mode, switch the processor to real mode, and branch to the appropriate piece of DOS system code.

In this chapter, I examine the process in detail to show the inner workings of the DPMI. I use three tools to assist the examination: a protected-mode Windows program called `pm.exe`, a real-mode DOS program called `rm.exe`, and a DOS device driver called `dd.sys`. The complete source code for these programs is available on the companion code disk.

INT 21h

`dd.asm` (Listing 28.1) shows partial code for a real-mode DOS device driver. This driver does not support an actual device. Instead, it simply reports (on the printer) which driver functions are called, plus any data that is exchanged among the driver and the calling program and the contents of the registers. `dpmi.h` (Listing 28.2), `dpmi.c` (Listing 28.3), and `pm.c` (Listing 28.4) contain much of the source code for a protected-mode Windows program that interacts with the real-mode device driver. `rm.c` (Listing 28.5) contains the source code to a real-mode DOS program that accesses the device driver. A good place to start exploring DPMI is by selecting Read/Write Driver from the menu of the Windows program (see Figure 28.1).



```
Int86x 6Bh
Selector: 2f7f Segment: 5a82
Registers
AX, BX, CX, DX: 1001, 1001, 1001, 1001
SI, DI: 1001, 1001
DS, ES: 2f7f, 2f7f
DPMI Int 6Bh
Selector: 2f7f Segment: 5a82
Registers
AX, BX, CX, DX: 1001, 1001, 1001, 1001
SI, DI: 1001, 1001
DS, ES: 2f7f, 2f7f
Get driver data
RM Pointer to driver data 2f7f:1001
PM pointer to driver data 306F:1001
```

Figure 28.1 `pm.exe` *in action.*

The first thing the Windows program does when you select the Read/Write Driver menu option is call `pmiCallDD()`. This function sets up a transfer buffer using the Windows API function `GlobalDosAlloc()` (see Figure 28.2). `pm.exe` will use the transfer buffer to exchange data with the device driver.

The low word of the return value from `GlobalDosAlloc()` contains a protected-mode selector that points to the transfer buffer. This selector is what the Windows program needs to access the buffer. The real-mode segment address corresponding to this selector is in the high word of `GlobalDOSAlloc()`'s return value The buffer is always paragraph-aligned, so the buffer offset is guaranteed to be zero. Furthermore, `GlobalDOSAlloc()` allocates the memory for the transfer buffer from the lower 1Mb of system memory. Thus it is accessible to real-mode programs as well as protected-mode programs.

```
/* Allocate the transfer buffer
 * Take selector from low word of return value
 * (real-mode segment is in high word)          */
if ((dwSegSelector = GlobalDosAlloc (BUFFER_SIZE))==NULL)
    return -1;

PMSelector = LOWORD (dwSegSelector);
RMSegment  = HIWORD (dwSegSelector);

/* Set pointer to transfer buffer */
lpTransferBuf = (LPSTR) MAKELONG (0, PMSelector);
```

Figure 28.2 *Allocating the transfer buffer.*

```
/* Read data from device driver */
inregs.x.ax = 0x3F00;     /* Function to read from device  */
inregs.x.bx = hDriver;    /* Handle to device driver       */
inregs.x.cx = 100;        /* Bytes to read                 */
sregs.ds =    PMSelector; /* Protected-mode address of
                           * transfer buffer               */
inregs.x.dx = 0;          /* Offset is 0 since buffer is
                           * paragraph aligned             */
/* Call DOS int 21 services */
intdosx (&inregs, &outregs, &sregs);
```

Figure 28.3 *Reading data from the device driver.*

It's important to understand that the device drivers and TSRs you load before running Windows all run in the same virtual 8086 machine (VM) that Windows does, the "system VM." This is why the memory that `GlobalDOSAlloc()` allocates is accessible to both the Windows program and the device driver. The memory containing these device drivers and TSRs in the system VM is virtually mapped to each other VM (or "DOS box") that Windows creates. So, although DOS programs that you spawn from Windows cannot directly access the memory allocated by `GlobalDOSAlloc()` (because they are in separate VMs), they can communicate with TSRs and device drivers loaded in the system VM and access indirectly through them.

Next, `pm.exe` opens the DOS device driver as a regular file (by calling `OpenFile()` with the device name) and sets the raw I/O mode. So far, no memory references have been made, so you haven't seen the DPMI in action. That's about to change. `pm.exe` next requests data from the driver. To do this it must pass the transfer buffer's address to the driver. `pm.exe` passes the address in registers `DS:DX`, where `DS` is assumed to be the segment of the buffer and `DX` the offset. `pm.exe` puts the protected-mode selector returned by `GlobalDOSAlloc()` into `DS` (note that filling a segment register with anything other than a valid selector will give a protection error in protected-mode Windows).

This doesn't seem right at first, because a protected-mode selector is meaningless to a real-mode program like the device driver, but something happens behind the scenes. Even though `pm.exe` filled `DS` with a protected-mode selector, the driver sees something else entirely. `pm.exe` displays the following text:

```
Read/Write Driver
Buffer selector:offset 08A5:0000 (segment 2df2)
This string is exported by the driver.
```

But the device driver prints this:

```
RMDRIVER: Opened
RMDRIVER: Received Read Request
RMDRIVER: Buffer segment:offset - 2DF2:0000
RMDRIVER: Received Write Request
RMDRIVER: Buffer segment:offset - 2DF2:0000
This string is from the application.
RMDRIVER: Closed
```

The precise values of the pointers will vary from the values shown when run on different machines.

The transfer buffer address is set to *08A5:0000* in the protected-mode application, but the driver receives an address of *2DF2:0000*. The DPMI intercepted the *INT 21h* and converted the protected-mode selector in the *DS* register to a real-mode segment. Obviously, the conversion is done correctly, since the Windows program and the device driver are able to share data in the form of a string.

Certain *INT 21h* functions are supported by the Windows DPMI. In fact, according to the Microsoft document *Windows INT 21h and NetBios Support for DPMI*, all the *INT 21h* services are supported except function *0h* (terminate program) and a subset of obsolete *File Control Block* functions. Also, since protected-mode uses its own interrupt table, functions *25h* and *35h* modify the protected-mode interrupt table and not the real-mode one. (Other minor details about the DPMI can be found in the official document.)

But what if the device driver uses an interrupt other than *21h*? Can you be certain the DPMI will convert memory references correctly and switch the processor to real-mode operation? The answer is yes, but be careful.

Other Interrupts

Here's the reason for caution: the Microsoft C interface to interrupts other than *INT 21h* is typically done through the *int86()* and *int86x()* functions. *int86()* is well behaved, but alas, the same cannot be said for *int86x()*, which you need in order to pass segment registers to the device driver. Compare the results of calling *int86x()* from a real-mode program and a protected-mode program. First, the real-mode program produces the following:

```
press <Enter> to read from driver
RMAPP: Buffer segment: offset - 2F0D:0016
This string is exported by the driver
Press <Enter> to write to driver
RMAPP: Buffer segment:offset - 2F0D:0016
Press <Enter> to attempt INT 6Bh
RMAPP: Registers
AX, BX, CX, DX: 1001, 1001, 1001, 1001
SI, DI: 1001, 1001
DS, ES: 1001, 1001
```

...while the device driver prints this:

```
RMDRIVER: Opened
RMDRIVER: Received Read Request
RMDRIVER: Buffer segment: offset - 2F0E:0006
RMDRIVER: Received Write Request
RMDRIVER: Buffer segment: offset - 2F0E:0006
RMDRIVER: This string is from the application.
RMDRIVER: Closed
RMDRIVER: Received INT 6bh
RMDRIVER: AX, BX, CX, DX: 1001, \ 1001, 1001, 1001
RMDRIVER: SI, DI: 1001, 1001
RMDRIVER: DS, ES: 1001, 1001
```

The `INT 6Bh` is the last thing the real-mode program does before terminating, and it results in a register dump by the device driver. The registers look the same in both the real-mode application and the device driver, as expected. But look at the results of the same `int86x()` call from the protected-mode Windows application (select `Int86x 6Bh` from the menu):

```
Int86x 6Bh
Buffer selector:offset 08A5:0000 (segment 34f4)
Registers
AX, BX, CX, DX: 1001, 1001, 1001, 1001
SI, DI: 1001, 1001
DS, ES: 8A5, 8A5
```

Afterwards, the device driver prints:

```
RMDRIVER: Received INT 6bh
RMDRIVER: AX, BX, CX, DX: 1001, 1001, 1001, 1001
RMDRIVER: SI, DI: 1001, 1001
RMDRIVER: DS, ES: 9273, C000
```

Look at registers `DS` and `ES`. Not only don't they resemble the real-mode segment corresponding to the protected-mode selector (as they did for the `INT 21h` calls), they don't even resemble the selector. In fact, they don't resemble anything at all. Apparently the protected-mode Windows implementation of `int86x()` is not fully implemented. The only way to pass segment registers (essential if pointers are exchanged) is to bypass the *int86x()* function and go straight to the DPMI. In *dpmi.c* (Listing 28.3) you will find an embedded assembly language function, *dpmiRMInt()*, which calls the DPMI directly.

This is one of many low-level DPMI functions that have found their way into common use. Using *dpmiRMInt()*, you can pass just about anything protected-mode and real-mode application might want to share — general registers, index registers, stack registers, segment registers, flags, stack values. Look at the results of using *dpmiRMInt()* instead of *int86x()* (select "*DPMI INT 6Bh*" from the menu of *pm.exe*):

```
DPMI Int 6Bh
Buffer selector:offset 08A5:0000 (segment 34f4)
Registers
AX, BX, CX, DX: 1001, 1001, 1001, 1001
SI, DI: 1001, 1001
DS, ES: 8A5, 8A5
```

Afterwards, the device driver prints:

```
RMDRIVER: Received INT 6bh
RMDRIVER: AX, BX, CX, DX: 1001, 1001, 1001, 1001
RMDRIVER: SI, DI: 1001, 1001
RMDRIVER: DS, ES: 08A5, 08A5
```

The segment registers passed correctly, but they contain the selector values, not the real-mode segments. Apparently, the DPMI isn't translating memory references as it did for the *INT 21h* calls above. This indicates that the segment value (from the low word of *GlobalDosAlloc()*), not the selector, should be placed in the registers.

Using the functions in *dpmi.c* (Listing 28.3), you can pass information, including pointers, in registers between protected-mode applications and real-mode device drivers.

Is that all there is to it? Not quite — there is still the issue of passing registers the other way, that is, from the device driver to the protected-mode program. As you will see, passing pointers the other way is trivial — it is using the pointers that creates headaches.

Getting Access to Foreign Segments

Run *pm.exe* and select "*Get Driver Data*" from the menu. This initiates the transfer of data from the device driver to the protected-mode Windows program. *dpmiRMInt()* issues an *INT 6Ah*, causing the device driver to load a pointer to its data area in registers *ES:BX*. *pm.exe* retrieves this pointer after the function returns. The trouble is, *pm.exe* cannot use the pointer. It is running in protected mode and can only access memory through protected-mode selectors, not via real-mode segment addresses. To access the memory that the segment address points to, *pm.exe* needs a protected-mode selector that refers to the same memory.

The high-level function *dpmiSegmentToSelector()* (in *dpmi.c*) does the dirty work required to give *pm.exe* access to the driver's data area. *dpmiSegmentToSelector()* first calls the low-level DPMI function *dpmiAllocNewSelector()* to create a new selector for the application. Then, using another low-level DPMI function, *dpmiReadDescriptor()*, it gets a descriptor (an entry in the Local Descriptor Table, which keeps track of the memory locations the program has access to). Using this descriptor as a guideline, it creates a new descriptor referencing the segment in the driver. A third low-level function, *dpmiSetDescriptor()*, is called to update the Local Descriptor Table with the new descriptor. The application can now use the selector returned by the function to form a pointer to the data in the device driver. If this sounds like a lot of protected-mode hocus-pocus, it is, but it is really not that complicated and it is the only way to obtain access permission to foreign data segments.

When you select "*Get Driver Data*", *pm.exe* displays:

```
Get Driver Data
RM Pointer to driver data AF1:875
PM Pointer to driver data 1055:0875
This data is contained in the device driver...
```

And the device driver prints:

```
RMDRIVER: Received INT 6ah
RMDRIVER: Pointer to local data 0AF1:0875
```

Conclusion

You can apply the procedures outlined in this article to interface protected-mode Windows programs with any DOS program resident in real memory before you run Windows, not just device drivers. TSRs are an obvious candidate. The listings include a sample device driver and two programs, one real-mode and one protected-mode, which interact with the driver. You will find these programs valuable as you explore ways to use your own device driver services from within protected-mode Windows programs. The example device driver is configured to print its output on LPT1. To direct output to another printer port, change the (`lptno`) to equate to the appropriate port and rebuild the driver. Be sure to include the line

```
DEVICE=<path>\DD.SYS
```

in your `config.sys` file, then reboot.

References

Duncan, Ray. Ed. *Extending DOS*. 2 ed. Reading, MA: Addison-Wesley 1992.

Listing 28.1 *A real-mode device driver.*

```
;INTERRUPT SERVICE ROUTINES
ioctl_out       PROC  NEAR              ;export ioctl control string to
                xor   ax, ax            ;application
                ret
ioctl_out       ENDP
io_read         PROC  NEAR              ;read data from device, place in
                jmp   SHORT ior01       ;buffer for application to read
ior_data        db    "This string is exported by the driver", 0
ior01:
                push  ds                ;save modified registers
                push  di
                push  bx
                push  cx
                printstr <"RMDRIVER: Received Read Request",13,10>
                mov   bx, 0             ;bx will be used to count the bytes ;read
                lds   di, DWORD PTR rheader.io_bufptr ;ds:di has address of
                                              ;transfer buffer
                printstr <"        ", "RMDRIVER: Buffer segment:offset - ">
                printh   ds
                printstr <":">
                printh   di
                printstr <13, 10>
ior02:          mov   cl, ior_data[bx]  ;next byte to read into buffer
                mov   BYTE PTR ds:[di+bx], cl    ;move next byte
                inc   bx                ;bump counter by one
                cmp   cl, 0             ;terminating null reached?
                jg    ior02             ;do next byte
                mov   WORD PTR rheader.io_count, bx   ;bytes transferred
                pop   cx                ;done - restore registers
                pop   bx
                pop   di
                pop   ds
                xor   ax, ax            ;return success (0)
                ret
io_read         ENDP
```

Listing 28.1 *continued*

```
io_write      PROC  NEAR           ;receives data from the application
              jmp   SHORT iow01    ;for writing to device
iow_data      db    100 DUP(0)     ;input buffer
iow01:
              push  ds             ;save machine state
              push  di
              push  bx
              push  cx
              printstr <"RMDRIVER: Received Write Request",13,10>
              mov   bx, 0          ;bx will count the bytes written
              lds   di, DWORD PTR rheader.io_bufptr ;ds:di has address of
                                              ;transfer buffer
              printstr <"             ", "RMDRIVER: Buffer segment:offset - ">
              printh  ds
              printstr <":">
              printh  di
              printstr <13, 10>
iow02:        mov   cl, BYTE PTR ds:[di+bx] ;write next byte
              mov   iow_data[bx], cl ;move next byte to buffer
              inc   bx             ;bump counter by one
              cmp   cl, 0          ;terminating null reached?
              mov   al, cl         ;character to print
              mov    ah, 0         ;print the character
              mov    dx, lptno     ;assumes port is available for printing
              int   17h            ;BIOS print routine invoked
              jg    iow02          ;do next byte
              mov   WORD PTR rheader.io_count, bx   ;bytes transferred
              printstr <13, 10>
              pop   cx             ;done - restore registers
              pop   bx
              pop   di
              pop   ds
              xor   ax, ax         ;return 0 - success
              ret
io_write      ENDP

dev_open      PROC  NEAR           ;device open
              printstr <"RMDRIVER: Opened",13,10>
              xor   ax, ax
              ret
dev_open      ENDP
dev_close     PROC  NEAR           ;device close
              printstr <"RMDRIVER: Closed",13,10>
              xor   ax, ax
              ret
dev_close     ENDP
```

Listing 28.1 *continued*

```
int6b_off       dw    ?                 ;segment of next int6b handler in chain
int6b_seg       dw    ?                 ;offset of next int6b handler in chain
int6bproc       PROC  FAR               ;int6b handler
                sti
                mov   cs:old_sp, sp ;set up local stack
                mov   cs:old_bp, bp ;save old bp
                mov   cs:old_ss, ss ;save old ss
                mov   bp, cs        ;set new bp, sp, ss
                mov   ss, bp
                mov   bp, OFFSET localstack
                mov   sp, OFFSET localstack

                ;print registers of interest
                printstr <"RMDRIVER: Received INT 6bh",13,10>
                printstr <"          ", "RMDRIVER: AX, BX, CX, DX: ">
                printh   ax
                printstr <", ">
                printh   bx
                printstr <", ">
                printh   cx
                printstr <", ">
                printh   dx
                printstr <13, 10>
                printstr <"          ", "RMDRIVER: SI, DI: ">
                printh   si
                printstr <", ">
                printh   di
                printstr <13, 10>
                printstr <"          ", "RMDRIVER: DS, ES: ">
                printh   ds
                printstr <", ">
                printh   es
                printstr <13, 10>
                mov   sp, cs:old_sp          ;restore system stack
                mov   bp, cs:old_bp
                mov   ss, cs:old_ss

                jmp   DWORD PTR int6b_off ;branch to default handler it will do iret
int6bproc       ENDP
```

Listing 28.1 *continued*

```
localdata       db    "This data is contained in the device-driver segment "
                db    "In order to access it, the application must perform "
                db    "an INT 6a and receive back a pointer.  The protected "
                db    "mode program must then convert the pointer to a "
                db    "selector and assume ownership of it.", 0
int6aproc       PROC  FAR                 ;intab handler
                sti
                mov   cs:old_sp, sp  ;set up local stack
                mov   cs:old_bp, bp  ;save old bp
                mov   cs:old_ss, ss  ;save old ss
                mov   bp, cs         ;set new bp, sp, ss
                mov   ss, bp
                mov   bp, OFFSET localstack
                mov   sp, OFFSET localstack

                mov   bx, cs
                mov   es, bx
                mov   bx, OFFSET localdata
                ;print pointer to local data
                printstr <"RMDRIVER: Received INT 6ah",13,10>
                printstr <"          ", "RMDRIVER: Pointer to local data ">
                printh   es
                printstr <":">
                printh   bx
                printstr <13, 10>
                mov   sp, cs:old_sp         ;restore system stack
                mov   bp, cs:old_bp
                mov   ss, cs:old_ss

                iret
int6aproc       ENDP

; End of File
```

Listing 28.2 *Header file for DPMI functions.*

```
/* DPMI.H */
#pragma pack(1)

#define SUCCESS      1
#define FAILURE      !SUCCESS

/* selector, segment, and offset types */
typedef WORD SELECTOR;
typedef WORD SEGMENT;
typedef WORD OFFSET;
/* Structure for making RM interrupts from 386 protected mode */
typedef struct {
    DWORD edi, esi, ebp, reserved, ebx, edx, ecx, eax;
    WORD flags, es, ds, fs, gs, ip, cs, sp, ss;
    } RM386_INT;

/* Structure of a protected mode descriptor */
typedef struct {
    WORD byte_limit, lo_addr;
    BYTE hi_addr, permissions, reserved, xhi_addr;
    } PM_DESCRIPTOR;

/* Low-level dpmi interface */
BOOL dpmiVersion (WORD *wFlags, WORD *wMajorVer, WORD *wMinorVer, WORD *wProcessor);
BOOL dpmiRMInt (WORD wIntNum, WORD wFlags, RM386_INT far *rmInt, WORD wStackVals);
SELECTOR dpmiAllocNewSelector (void);
BOOL dpmiSetDescriptor (SELECTOR slSelector, PM_DESCRIPTOR far *pmDescriptor);
BOOL dpmiReadDescriptor (SELECTOR slSelector, PM_DESCRIPTOR far *pmDescriptor);
BOOL dpmiFreeSelector (SELECTOR slSelector);

/* High level dpmi */
BOOL dpmiSegmentToSelector (SEGMENT sgSegment, DWORD dwBytes, SELECTOR *slSelector);
void vSetRMDescriptorAddr (SEGMENT sgSeg, PM_DESCRIPTOR *pmdD);

DWORD FAR PASCAL GlobalDosAlloc (DWORD);
WORD  FAR PASCAL  GlobalDosFree  (SELECTOR);

/* End of File */
```

Listing 28.3 *Routines to access DPMI.*

```
#include <dos.h>
#include <windows.h>
#include "dpmi.h"

//Simulate real mode interrupt from protected mode Windows
BOOL dpmiRMInt(WORD wIntno, WORD wFlags, RM386_INT far *rmInt, WORD wStackVals)
{
    if (wFlags)
    {
        wIntno |= 0x100;
    } //End if (flags set)

    _asm
    {
        push di
        mov  ax, 0300h
        mov  bx, word ptr wIntno
        mov  cx, word ptr wStackVals
        les  di, dword ptr rmInt
        int  31h
        jc   error
        mov  ax, 1
        jmp  short done
    } //End (low-level dpmi real-mode interrupt)
error:
    _asm xor ax, ax
done:
    _asm pop di

} //End function (dpmiRMInt)

//Get the DPMI version number, processor type
BOOL dpmiVersion (WORD *wFlags, WORD *wMajor, WORD *wMinor, WORD *wProcessor)
{
BYTE bMaj, bMin, bProc;
WORD wFl;
BOOL bRval;

    _asm
    {
        mov ax, 1686h
        int 2fh
        not ax
        mov bRval, ax
    } //End (determine if dpmi is present)
    if (!bRval)
    {
        return FALSE;
    } //End if (dpmi not present)
```

Listing 28.3 *continued*

```
    _asm
    {
        mov ax, 0400h
        int 31h
        mov bProc,  cl
        mov wFl, bx
        mov bMaj, ah
        mov bMin, al
    } //End (get version information)
    *wProcessor = bProc;
    *wFlags     = wFl;
    *wMinor     = bMin;
    *wMajor     = bMaj;

     return TRUE;
} //End function (dpmiVersion)

//Allocate a single LDT selector
SELECTOR dpmiAllocNewSelector (void)
{
    _asm
    {
        xor ax, ax
        mov cx, 1
        int 31h
        jc  error
        jmp short done
    } //End (get selector)
error:
    _asm xor ax, ax
done:;
} //End function (dpmiAllocNewSelector)

//Assign new descriptor to our application
BOOL dpmiSetDescriptor (SELECTOR slSelector, PM_DESCRIPTOR far *pmDescriptor)
{
    _asm
    {
        mov  bx, WORD PTR slSelector
        mov  ax, 000ch
        push di
        les  di, DWORD PTR pmDescriptor
        int  31h
        jc   error
        mov  ax, 1
        jmp  short done
    } //End (set descriptor)
error:
    _asm xor ax, ax
done:
    _asm pop di
} //End function (dpmiSetDescriptor)
```

Listing 28.3 *continued*

```
//Read a descriptor from our application
BOOL dpmiReadDescriptor (SELECTOR slSelector, PM_DESCRIPTOR far *pmDescriptor)
{
    _asm
    {
        mov  bx, WORD PTR slSelector
        mov  ax, 000bh
        push di
        les  di, DWORD PTR pmDescriptor
        int  31h
        jc   error
        mov  ax, 1
        jmp  short done
    } //End (read descriptor)
error:
    _asm xor ax, ax
done:
    _asm pop di
} //End function (dpmiReadDescriptor)

//Free an LDT selector
BOOL dpmiFreeSelector (SELECTOR slSelector)
{
    _asm
    {
        mov ax, 0001h
        mov bx, WORD PTR slSelector
        int 31h
        jc  error
        mov ax,1
        jmp short done
    } //End (free selector)
error:
    _asm xor ax, ax
done:;
} //End function (dpmiFreeSelector)
```

Listing 28.3 *continued*

```
/*****************************************************************************

    FUNCTION:  dpmiSegmentToSelector

    PURPOSE:  Converts a real-mode segment to a protected-mode selector,
              and gives the application access to that selector.
    COMMENTS:

*****************************************************************************/
BOOL dpmiSegmentToSelector (SEGMENT sgSegment, DWORD dwBytes, SELECTOR *slSelector)
{
PM_DESCRIPTOR pmdDescriptor;
LPSTR lpPointDS = (LPSTR) &pmdDescriptor;
char szMessage[528];

    //Allocate new selector (with permissions belonging to this application)
    if ((*slSelector = dpmiAllocNewSelector())==0)
    {
        return FAILURE;
    } //End if (could not allocate new selector)
    //Get descriptor for data segment this program is using
    //FP_SEG returns the selector of this program's data segment
    //when applied to any far pointer in the program.
    if (!dpmiReadDescriptor(FP_SEG(lpPointDS), &pmdDescriptor))
    {
        return FAILURE;
    } //End if (could not read descriptor for program's data segment)
    //Use the same permissions for new descriptor.  Modify limit
    //and address fields.
    pmdDescriptor.byte_limit = dwBytes -1;
    //Set lo, hi, and xhi addresses in the screwy 32 bit descriptor format
    vSetRMDescriptorAddr (sgSegment, &pmdDescriptor);
    if (!dpmiSetDescriptor (*slSelector, &pmdDescriptor))
    {
        return FAILURE;
    } //End if (could not set new descriptor)
    return SUCCESS;
} //End function (dpmiRealToSelector)
```

Listing 28.3 *continued*

```
void vSetRMDescriptorAddr (SEGMENT sgSeg, PM_DESCRIPTOR *pmdD)
{
DWORD  dw32bitOffset;
    //Offsets in 32 bit 386 mode are 32 bits.  Real-mode segments
    //are 16 bit paragraph addresses, which correlate to 20bit
    //memory addresses in the lower 1MB range by performing a
    //4 bit left shift.
    dw32bitOffset = ((DWORD) sgSeg) << 4;
    //Now for the screwy descriptor format:
    //lo address is lower WORD of 32 bit offset
    pmdD->lo_addr = (WORD) dw32bitOffset;
    //hi address is anything in excess of 16 bits
    pmdD->hi_addr = (BYTE) (dw32bitOffset >> 16);
    //Can't get at memory addressed by this byte in real mode
    pmdD->xhi_addr = (BYTE) 0;

} //End function (vSetRMDescriptorAddr)

/* End of File */
```

Listing 28.4 *A protected-mode Windows program.*

```
long FAR PASCAL MainWndProc(HWND hWnd, unsigned message, WORD wParam, LONG lParam)
{
HDC  hDC;

    switch (message)
    {
        case WM_CREATE:
        {
        WORD wFlags=0, wMajorVer=0, wMinorVer=0, wProcessor=0;
            if ( !(GetWinFlags() & WF_PMODE) ||
                !dpmiVersion (&wFlags, &wMajorVer, &wMinorVer, &wProcessor))
            {
                MessageBox(hWnd,"DPMI services unavailable","PMAPP",MB_OK);
                PostQuitMessage(0);
            }  //End if (not protected mode, or dpmi not available)
            else
            {
            char szPrompt[80];
                if (wProcessor==2) wProcessor=286;
                if (wProcessor==3) wProcessor=386;
                if (wProcessor==4) wProcessor=486;
                if (GetWinFlags()&WF_STANDARD)
                {
                    sprintf (szPrompt, "Standard Mode DPMI ver. %x.%x: Processor %d",
                             wMajorVer, wMinorVer, wProcessor);
                } //End if (standard mode)
                else
                {
                    sprintf (szPrompt, "Enhanced Mode DPMI ver. %d.%d: Processor %d",
                             wMajorVer, wMinorVer, wProcessor);
                } //End if (enhanced mode)
                MessageBox(hWnd, szPrompt,
                "PMAPP",MB_OK);
            } //End if (protected mode and dpmi available)
            break;
        } //End case (WM_CREATE)
```

Listing 28.4 *continued*

```
        case WM_COMMAND:           /* message: command from application menu */
            switch (wParam)
              {
              case RW:
                  pmiCallDD (hDC = GetDC(hWnd));
                  ReleaseDC (hWnd, hDC);
                  break;
              case INT86:
                  pmiInt6b (hDC = GetDC(hWnd));
                  ReleaseDC (hWnd, hDC);
                  break;
              case DPMI:
                  pmiDPMIInt6b (hDC = GetDC(hWnd));
                  ReleaseDC (hWnd, hDC);
                  break;
              case DDATA:
                  pmiGetDriverData (hDC = GetDC(hWnd));
                  ReleaseDC (hWnd, hDC);
                  break;
              default:                            /* Lets Windows process it */
                return (DefWindowProc(hWnd, message, wParam, lParam));
                break;
              }
            break;
        case WM_DESTROY:                  /* message: window being destroyed */
            PostQuitMessage(0);
            break;

        default:                          /* Passes it on if unproccessed     */
            return (DefWindowProc(hWnd, message, wParam, lParam));
    }
    return (NULL);
}
```

Listing 28.4 *continued*

```
/*******************************************************
    FUNCTION:  pmiCallDD
    PURPOSE:   Calls real-mode device driver, swaps data
               using a transfer buffer
*******************************************************/
int pmiCallDD(HDC hDC)
{
LPSTR    lpTransferBuf;                  //points to transfer buffer
HANDLE   hDriver;                        //handle to driver
OFSTRUCT OfFileStruct;                   //structure for opening file

    sprintf (szMessage, "%s", "Read/Write Driver");
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    // Allocate the transfer buffer
    // Take selector from low word of return value
    // (real-mode segment is in high word)
    if ((dwSegSelector = GlobalDosAlloc (BUFFER_SIZE))==NULL)
    {
        return -1;
    } //End if (could not allocate transfer buf)
    PMSelector = LOWORD (dwSegSelector);
    RMSegment  = HIWORD (dwSegSelector);
    // Set pointer to transfer buffer
    lpTransferBuf = (LPSTR) MAKELONG (0, PMSelector);

    // Get a handle to the device driver
    if ((hDriver = OpenFile ("RMDRIVER", &OfFileStruct, OF_READWRITE))==-1)
    {
        sprintf (szMessage, "Error opening driver");
        TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
        return -1;
    } //End if (error opening driver)
```

Listing 28.4 *continued*

```
    //Set raw io mode
    pmiSetRawMode (hDriver);

    // Read data from device driver
    inregs.x.ax = 0x3F00;     //Function to read from device
    inregs.x.bx = hDriver;    //Handle to device driver
    inregs.x.cx = 100;        //Bytes to read
    sregs.ds =   PMSelector;  //Protected mode address of transfer buffer
    inregs.x.dx = 0;          //Offset is 0 since buffer is paragraph aligned
    sprintf (szMessage, "Buffer selector:offset %Fp  (segment %x)", lpTransferBuf, RMSegment);
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    //Call DOS int 21 services
    intdosx (&inregs, &outregs, &sregs);

    // Print the buffer; it now contains data from the device driver.
    lstrcpy ((LPSTR) szMessage, lpTransferBuf);
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));

    // Write string to device driver
    lstrcpy (lpTransferBuf, (LPSTR) szOutputBuf);
    inregs.x.ax = 0x4000;     //Function to write to device
    inregs.x.bx = hDriver;    //Handle to the driver
    inregs.x.cx = 100;        //Bytes to write
    sregs.ds =   PMSelector;  //Protected mode address of transfer buffer
    inregs.x.dx = 0;          //Offset is 0 since buffer is paragraph aligned
    intdosx (&inregs, &outregs, &sregs);

    GlobalDosFree (PMSelector);

    // Close the device driver
    if (_lclose (hDriver)!=0)
    {
        sprintf (szMessage, "Error closing driver");
        TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
        return -1;
    } //End if (error closing driver)

    return 0;
} //End function (pmiCallDD)
```

Listing 28.4 *continued*

```
/*********************************************************
    FUNCTION:  pmiInt6b
    PURPOSE:  Performs int6b using int86x
*********************************************************/
int pmiInt6b(HDC hDC)
{
    sprintf (szMessage, "%s", "Int86x 6Bh");
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    // Allocate the transfer buffer
    // Take selector from low word of return value
    // (real-mode segment is in high word)
    if ((dwSegSelector = GlobalDosAlloc (BUFFER_SIZE))==NULL)
    {
        return -1;
    } //End if (could not allocate transfer buf)
    PMSelector = LOWORD (dwSegSelector);
    RMSegment  = HIWORD (dwSegSelector);
    sprintf (szMessage, "Selector: %x Segment: %x", PMSelector, RMSegment);
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));

    inregs.x.ax =  0x1001;
    inregs.x.bx =  0x1001;
    inregs.x.cx =  0x1001;
    inregs.x.dx =  0x1001;
    inregs.x.si =  0x1001;
    inregs.x.di =  0x1001;
    sregs.ds =     PMSelector;
    sregs.es =     PMSelector;
    sprintf (szMessage, "Registers");
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    sprintf (szMessage, "AX, BX, CX, DX: %x, %x, %x, %x", inregs.x.ax,
             inregs.x.bx, inregs.x.cx, inregs.x.dx);
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    sprintf (szMessage, "SI, DI: %x, %x", inregs.x.si, inregs.x.di);
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    sprintf (szMessage, "DS, ES: %x, %x", sregs.ds, sregs.es);
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    int86x (0x6b, &inregs, &outregs, &sregs);

    GlobalDosFree (PMSelector);

    return 0;
} //End function (pmiInt6b)
```

Listing 28.4 *continued*

```
/******************************************************
    FUNCTION:  pmiSetRawMode
    PURPOSE:  Places the device driver in raw (binary) mode
*******************************************************/
int pmiSetRawMode(HANDLE hDriver)
{
    // Set the dd to raw mode - get, then set device control bits
    //bx - file handle for the driver
    //ax - 0x4400  Function for IOCTL call to read device control bits
    inregs.x.bx = hDriver;
    inregs.x.ax = 0x4400;
    intdos (&inregs, &outregs);
    // Now set raw mode bit of the device control bits
    //dl - bit 0x20 is set to indicate DOS should use raw mode with this
    //     device.  All other bits are unchanged.
    //ax - 0x4401   IOCTL function to set device bits
    inregs.x.bx = hDriver;
    inregs.h.dh=0;
    inregs.h.dl= outregs.h.dl | 0x20;
    inregs.x.ax = 0x4401;
    intdos (&inregs, &outregs);

    return 0;
} //End function (pmiSetRawMode)
/******************************************************
    FUNCTION:  pmiDPMIInt6b
    PURPOSE:  Performs int6b using DPMI
*******************************************************/
int pmiDPMIInt6b(HDC hDC)
{
    sprintf (szMessage, "%s", "DPMI Int 6Bh");
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    // Allocate the transfer buffer
    // Take selector from low word of return value
    // (real-mode segment is in high word)
    if ((dwSegSelector = GlobalDosAlloc (BUFFER_SIZE))==NULL)
    {
        return -1;
    } //End if (could not allocate transfer buf)
    PMSelector = LOWORD (dwSegSelector);
    RMSegment  = HIWORD (dwSegSelector);
    sprintf (szMessage, "Selector: %x Segment: %x", PMSelector, RMSegment);
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
```

Listing 28.4 *continued*

```
    rmInt.edi = 0x1001;
    rmInt.esi = 0x1001;
    rmInt.ebx = 0x1001;
    rmInt.edx = 0x1001;
    rmInt.ecx = 0x1001;
    rmInt.eax = 0x1001;
    rmInt.es =  PMSelector;
    rmInt.ds =  PMSelector;
    sprintf (szMessage, "Registers");
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    sprintf (szMessage, "AX, BX, CX, DX: %lx, %lx, %lx, %lx", rmInt.eax,
             rmInt.ebx, rmInt.ecx, rmInt.edx);
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    sprintf (szMessage, "SI, DI: %lx, %lx", rmInt.esi, rmInt.edi);
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    sprintf (szMessage, "DS, ES: %x, %x", rmInt.ds, rmInt.es);
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    dpmiRMInt (0x6b, 0, &rmInt, 0);

    GlobalDosFree (PMSelector);

    return 0;
} //End function (pmiDPMIInt6b)
/*******************************************************
    FUNCTION:  pmiGetDriverData
    PURPOSE:  Retrieves data from the device driver segment
*******************************************************/
int  pmiGetDriverData(HDC hDC)
{
LPSTR lpDriverData;
LPSTR lpDriverData2;
    sprintf (szMessage, "%s", "Get driver data");
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    dpmiRMInt (0x6a, 0, &rmInt, 0);
    sprintf (szMessage, "RM Pointer to driver data %x:%x", rmInt.es, mInt.ebx);
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    if (!dpmiSegmentToSelector(rmInt.es, rmInt.ebx + 500, &PMSelector))
    {
        return FAILURE;
    }  //End if (error getting access to driver segment)
    lpDriverData = (LPSTR) MAKELONG (rmInt.ebx, PMSelector);
    sprintf (szMessage, "PM pointer to driver data %Fp", lpDriverData);
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) szMessage, strlen(szMessage));
    TextOut (hDC, 5, pmTextY+=TEXT_INC, (LPSTR) lpDriverData, lstrlen(lpDriverData));
    //Always free selectors when done with them
    dpmiFreeSelector (PMSelector);
    return SUCCESS;
} //End (pmiGetDriverData)

/* End of File */
```

Listing 28.5 *A real-mode DOS program.*

```
/* Interacts with the real-mode device driver */
#include <dos.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <malloc.h>

char szAppString[]= "RMDRIVER: This string is from the application.";
union REGS inregs, outregs;
struct SREGS sregs;

main()
{
char far *lpTransferBuf;
unsigned hDriver;
char szMessage[100];
int i;

    //Allocate the transfer buffer
    if ((lpTransferBuf = (char far *) _fmalloc (100))==NULL)
    {
        printf ("RMAPP: Error allocating buffer\n");
        exit (1);
    } //End if (could not allocate transfer buf)

    //Get a handle to the device driver
    if (_dos_open ("RMDRIVER", O_RDWR, &hDriver)!=0)
    {
        printf ("RMAPP: Error %d opening driver (%d)\n", errno, ENOENT);
        exit (1);
    } //End if (error opening driver)
    //Set raw io mode
    rmiSetRawMode (hDriver);

    fflush (stdin);
    printf ("Press  <Enter> to read from driver\n");
    while (!kbhit());
    getchar();

    //Read from device driver
    inregs.x.ax = 0x3F00;       //Read from device
    inregs.x.bx = hDriver;      //handle to the driver
    inregs.x.cx = 100;          //maximum bytes to read
    sregs.ds =     FP_SEG (lpTransferBuf);       //segment of transfer buffer
    inregs.x.dx = FP_OFF (lpTransferBuf);       //offset of transfer buffer
    printf ("RMAPP: Buffer segment:offset - %Fp\n", lpTransferBuf);
    intdosx (&inregs, &outregs, &sregs);
```

Listing 28.5 *continued*

```
    // Print the buffer;  it now contains data from the device
    //driver.
    i=0;
    while (szMessage[i]= lpTransferBuf[i]) i++;
    printf ("%s\n", szMessage);

    fflush (stdin);
    printf ("Press  <Enter> to write to driver\n");
    while (!kbhit());
    getchar();

    // Write string to device driver
    i=0;
    while (lpTransferBuf[i++]= szAppString[i]);
    inregs.x.ax = 0x4000;       //Write to device
    inregs.x.bx = hDriver;      //Handle to driver
    inregs.x.cx = 100;          //Bytes to write
    sregs.ds =    FP_SEG (lpTransferBuf);   //segment of transfer buffer
    inregs.x.dx = FP_OFF (lpTransferBuf);   //offset of transfer buffer
    printf ("RMAPP: Buffer segment:offset - %Fp\n", lpTransferBuf);
    intdosx (&inregs, &outregs, &sregs);

    if (lpTransferBuf) _ffree (lpTransferBuf);

    //Close the device driver
    if (_dos_close (hDriver)!=0)
    {
        printf ("RMAPP: Error %d closing driver (%d)\n", errno, ENOENT);
        exit (1);
    } //End if (error closing driver)

    fflush (stdin);
    printf ("Press  <Enter> to attempt INT 6Bh\n");
    while (!kbhit());
    getchar();

    inregs.x.ax =  0x1001;
    inregs.x.bx =  0x1001;
    inregs.x.cx =  0x1001;
    inregs.x.dx =  0x1001;
    inregs.x.si =  0x1001;
    inregs.x.di =  0x1001;
    sregs.ds =     FP_SEG(lpTransferBuf);
    sregs.es =     FP_SEG(lpTransferBuf);
    printf ("RMAPP: Registers\n");
    printf ("\tRMAPP: AX, BX, CX, DX: %x, %x, %x, %x\n", inregs.x.ax,
             inregs.x.bx, inregs.x.cx, inregs.x.dx);
    printf ("\tRMAPP: SI, DI: %x, %x\n", inregs.x.si, inregs.x.di);
    printf ("\tRMAPP: DS, ES: %x, %x\n", sregs.ds, sregs.es);
    int86x (0x6b, &inregs, &outregs, &sregs);

} //End function (main)
```

Listing 28.5 *continued*

```
/****************************************************************************

    FUNCTION:  rmiSetRawMode

    PURPOSE:  Places the device driver in raw (binary) mode

    COMMENTS:


****************************************************************************/
int rmiSetRawMode(unsigned hDriver)
{
    // Set the dd to raw mode - get, then set device control bits
    //bx - file handle for the driver
    //ax - 0x4400  Function for IOCTL call to read device control bits
    inregs.x.bx = hDriver;
    inregs.x.ax = 0x4400;
    intdos (&inregs, &outregs);
    // Now set raw mode bit of the device control bits
    //dl - bit 0x20 is set to indicate DOS should use raw mode with this
    //     device.  All other bits are unchanged.
    //ax - 0x4401   IOCTL function to set device bits
    inregs.x.bx = hDriver;
    inregs.h.dh=0;
    inregs.h.dl= outregs.h.dl | 0x20;
    inregs.x.ax = 0x4401;
    intdos (&inregs, &outregs);

    return 0;
} //End function (rmiSetRawMode)

/* End of File */
```

Chapter 29

DDE to DOS

Guy Eddon

DDE is an interprocess communication method that lets Windows applications exchange data. For example, WinWord macros can access data in an Excel spreadsheet and insert it into a document. One important source of data is inaccessible to DDE: DOS programs. Although Windows does not offer a DDE interface to DOS programs, you can construct your own interface using a Virtual Device Driver (VxD) to bridge the gap between DOS and Windows. This article shows how to use a VxD and an auxiliary Windows program to let a DOS program act as a DDE server for Windows programs.

The code in this article assumes you are using enhanced-mode Windows v3.1. To build the programs provided with this article, you will need the Microsoft Windows Device Driver Kit (DDK).

What Is DDE?

Dynamic Data Exchange (DDE) is a message-based protocol that lets two Windows applications communicate. The DDE protocol is based on a client/server model. In this model, the client initiates the DDE conversation, and then requests data from the server. The server in turn responds with the requested data if it is available.

DDE is built on the Windows' messaging system. Two applications communicate by posting special DDE messages to one another, using hidden windows reserved for this purpose. Table 29.1 lists the DDE messages with their corresponding functions.

DDE conversations are called links and they fall into three categories: cold links, warm links, and hot links. The type of link indicates the degree of involvement between the client and the server. A cold link occurs when DDE-aware applications exchange data once. If the client should need more data it must initiate another DDE conversation. With a warm link, the server informs the client application whenever new data on the topic being discussed arrives. A hot link indicates that the server will send the client all updated data whenever it arrives.

DDE has been available since the first release of Microsoft Windows v1.0, and is now an integral part of Windows. As is true of other Windows

Message	Description
`WM_DDE_ACK`	Acknowledges a received DDE message
`WM_DDE_ADVISE`	Establishes a warm or hot link
`WM_DDE_DATA`	Sends data
`WM_DDE_EXECUTE`	Sent to the client with a string for execution
`WM_DDE_INITIATE`	Used to start a DDE conversation
`WM_DDE_POKE`	Used by the client to poke data to the server
`WM_DDE_REQUEST`	Posted by the client to request data
`WM_DDE_TERMINATE`	Used to end a DDE conversation
`WM_DDE_UNADVISE`	Terminates a warm or hot link

Table 29.1 *The DDE messages.*

components, DDE has been improved and refined in successive releases. Its latest incarnation, as the Dynamic Data Exchange Management Library (DDEML), represents a major improvement over the original, which was generally regarded as fit for use only by super programmers. The perceived difficulty can be attributed to the fact that since DDE itself is a simple message-based protocol, it provides none of the convenient high-level functions that empower a programmer with control. DDEML addresses this by providing a high-level API that manages all of the DDE messages for you. Since DDEML functions are implemented using DDE, applications written to the DDEML API are compatible with applications that use DDE directly.

The System VM

The key to DOS compatibility in enhanced-mode Windows is the 80386's ability to supply virtual 8086 machines: each virtual machine believes its address space starts with the crucial 640Kb of conventional memory. When WIN386 loads the Virtual Machine Manager (VMM), the VMM in turn creates the System VM. Later, the VMM manages the creation and termination of all other virtual machines. All Windows applications run in the System VM, which is basically a protected-mode DOS box, and this means that all Windows applications share the same address space. So once the System VM is set up, Windows is loaded as a protected-mode DOS application running inside a V86-mode box. The System VM is the most important VM because it exists from the time Windows starts to the time it exits.

Figure 29.1 shows an overview of Windows and the DOS-to-DDE connection. As noted earlier, all the Windows applications run within the System VM. The original copy of DOS, loaded when you booted your computer, now becomes a small part of the Windows operating system. The VMM is responsible for maintaining the System VM, and the creation and termination of other VMs at the user's request. These secondary VMs run DOS and consist of copies of the original DOS contained within the System VM. The virtual devices at the top of Figure 29.1 represent the 32-bit protected-mode part of Windows. This protected subsystem provides the base DOS application compatibility layer in Windows.

Preemptive Multitasking

Enhanced-mode Windows v3.x can run several DOS applications and Windows applications at the same time. It does so through virtual machines which pretend to be in real mode so that DOS applications will run. The common perception that OS/2 supports preemptive multitasking while Windows does not is thus incorrect: Windows, in fact, supports a preemptive multitasking mechanism among all virtual machines on the system.

The VMM is capable of preemptive switching among all virtual machines in the system. Since all Windows applications run in one virtual

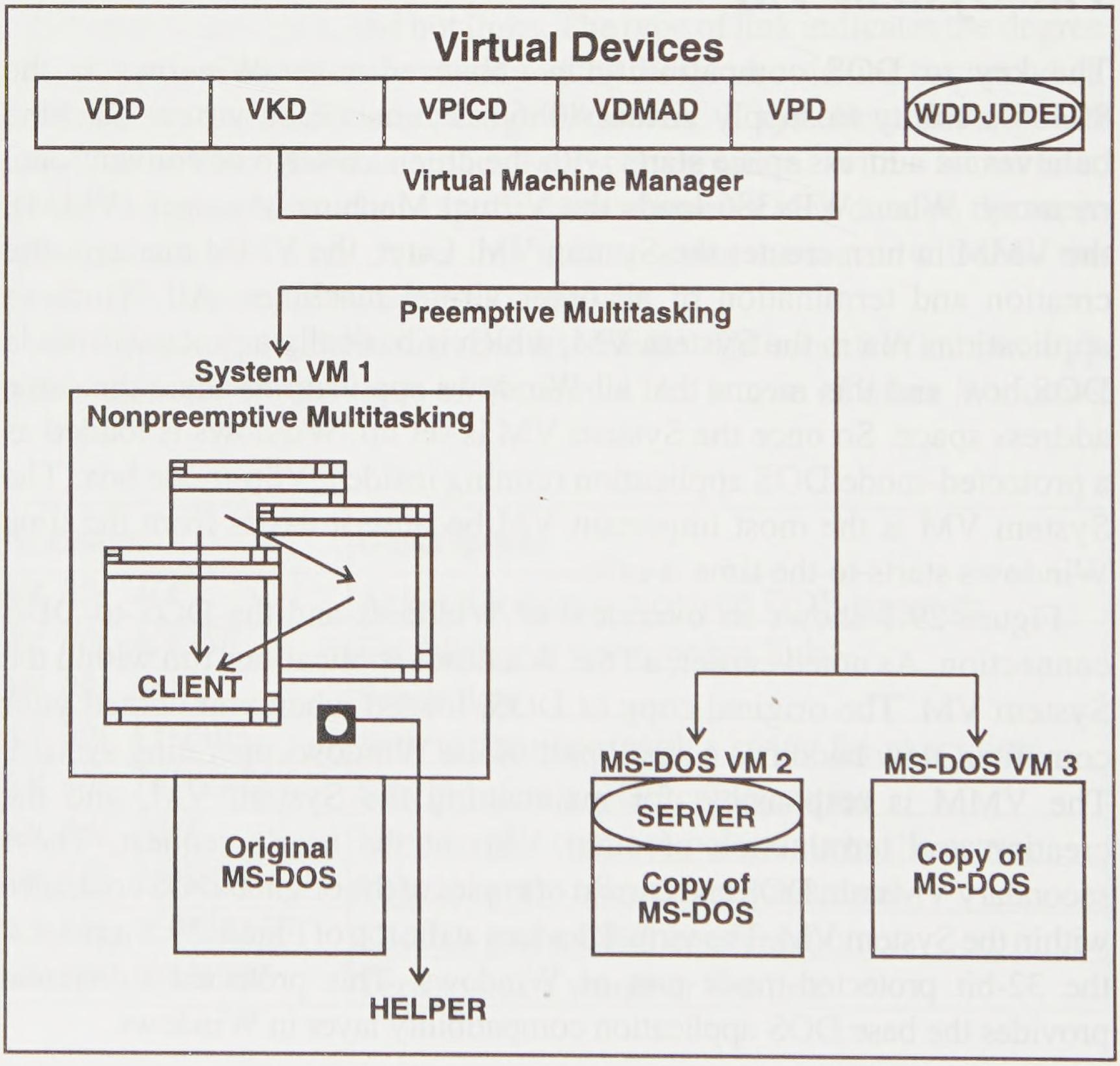

Virtual Devices
VDD
VKD
VPICD
VDMAD
VPD
WDDJDDED
Virtual Machine Manager
Preemptive Multitasking
System VM 1
Nonpreemptive Multitasking
CLIENT
Original MS-DOS
HELPER
MS-DOS VM 2
SERVER
Copy of MS-DOS
MS-DOS VM 3
Copy of MS-DOS

Figure 29.1 *Windows system architecture.*

machine — the System VM — Windows programs are non-preemptively multitasked by a message-based multitasker. Each Windows application has a message queue which it must check in a timely manner if the system is to continue operating properly.

What Is a VxD?

Virtual Device Drivers (VxDs) let Windows virtualize devices, so that both Windows and each separate DOS session can view itself as the exclusive owner of devices such as the screen. However, VxDs also provide an extensive API beyond the standard Windows API. I use a VxD as a logical bridge to move data between a Windows application and a DOS application.

A VxD is a special type of Windows program that assists with DOS application compatibility. Enhanced-mode Windows uses virtual devices to virtualize hardware so that multiple DOS applications will not be corrupted by simultaneous attempts to access the same hardware. This is necessary because DOS applications assume comprehensive access to the system's resources. By this I mean that the average DOS program believes it can go anywhere and do anything. However, in Windows this may no longer be the case, as multiple DOS programs may be running simultaneously. Thus, Windows must trick DOS applications into believing that they are executing alone, when in fact they are not. The VxD model implements this capability, and Windows v3.1 comes with several VxDs that virtualize access to most of the standard PC hardware.

The most obvious example of hardware virtualization is the Virtual Display Device (VDD). Nearly all commercial DOS applications perform video output by writing directly to video memory. Windows cannot tolerate this approach, because multiple DOS and Windows applications would overwrite one another's display. This is where the VDD comes into play. Each DOS application is led to believe it is writing directly to, and reading directly from, video memory; in fact, though, the VDD causes the DOS application to write its display to a memory buffer, then updates the DOS window based upon this buffer. This sequence of events permits multiple DOS and Windows applications to coexist peacefully.

The VxD's ability to virtualize hardware also make it useful for implementing a communication path between DOS and Windows applications. I have created a VxD (`wddjdded.386`) that handles the low-level problems of passing data efficiently from a DOS application to the Windows environment.

How It Works

Since DDE conversations are based on the ability to send and receive Windows messages, only Windows applications can participate. Windows simply does not provide any mechanism which allows a DOS program to do this. Figure 29.2 shows how I use a VxD in conjunction with a "helper" Windows application to provide a DDE connection between Windows and DOS.

The DDE client shown in Figure 29.2 doesn't know that the DOS application has no legal role in their relationship. The client application initiates a perfectly legal conversation with the Windows DDE helper application.

The helper notifies the virtual DDE device (`wddjdded.386`) of the messages it is receiving. Once the DOS DDE server receives the message, it tells the VxD how to respond, and the message is passed back through the chain of command to the Windows DDE helper application and from there to the client application via a regular DDE conversation. Whenever data must be passed from the DOS DDE server to the client, the virtual DDE device copies the data from the server's data buffer to the helper's data buffer. From there the data is transferred to the DDE client via normal DDE messages, without help from the virtual device.

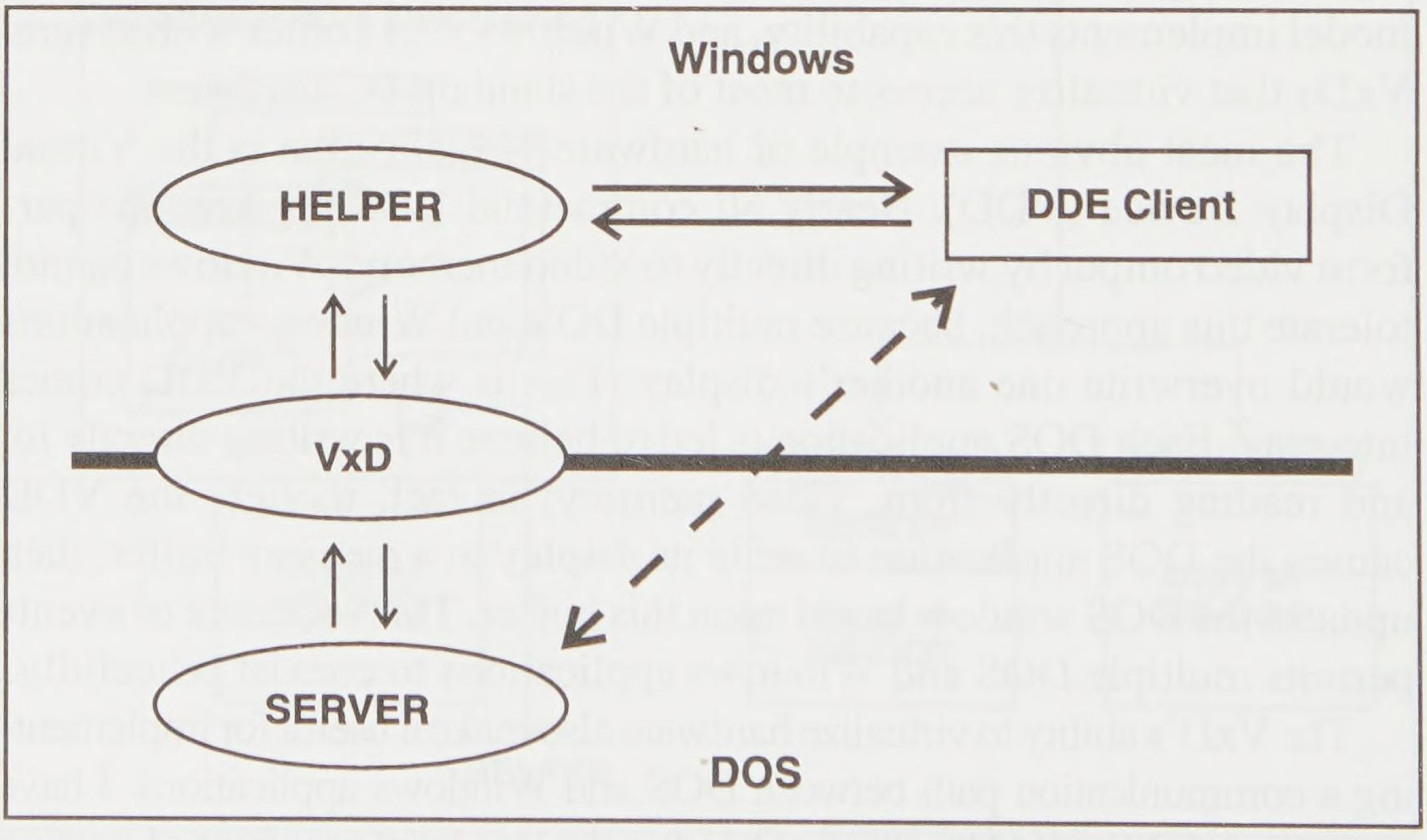


Figure 29.2 *The DDE to DOS model.*

In my example, the topic of the DDE conversation is "Amount." The client application can be any DDE-enabled Windows application, including Microsoft Word or Excel. The DOS server then tells the helper application how to respond to the client.

The DOS Server Program

As noted earlier, the server in the DDE conversation is a DOS application. Therefore, the client can be any Windows application which requires data from the server. *server.c* (Listing 29.1) is the example DOS server code that provides data to any client Windows DDE application. It also contains a set of functions which write directly to video memory, because Standard C library functions such as *printf()* leave the system in an unstable state when called from a VxD.

The Windows Helper Program

The Windows DDE helper application is called *helper.exe*. *helper.exe* is the Windows application that actually carries on the DDE conversation on behalf of the DOS application. Since *helper.exe* does all the actual DDE message passing with the client, it must also inform the DOS application of information received from the client. *helper.exe* communicates with the DOS application by passing the DDE messages through *wddjdded.386*. The complete source code for the help application (*helper.c*, *helper.rc*, *helper.ico*, *helper.def*, and *helper.h*) is available on the companion code disk.

wddjdded.386

The *wddjdded.386* VxD virtualizes DDE messages for use by a DOS application. It takes instructions from the DOS application and passes them to the Windows helper application. It also returns messages from the helper application to the DOS server application. *wddjdded.asm* (Listing 29.2) contains the main source code for the VxD, and *wddjded.def* (Listing 29.3) contains its module definition file.

The DDE Client

The idea for the example in this article came from the DDE example in the Windows v3.1 Software Development Kit (SDK), located in the `c:\c700\guide\dde` directory. In this directory you will find two Windows applications: client and server. The DOS DDE server in this article duplicates the functionality of the Windows server. I recommend that for testing purposes you run the Windows client with the DOS server in this article. For a step-by-step description of how to test the example provided with this article, see Table 29.2.

1	Place all the files in `C:\VDDED`.
2	Add ’`device=c:\vdded\wddjdded.386`’ to the `[386Enh]` section of your `C:\WINDOWS\SYSTEM.INI` file.
3	Load Windows.
4	Choose File-Run from the Program Manager, and type `C:\VDDED\HELPER`.
5	Choose File-Run from the Program Manager, and type `C:\VDDED\CLIENT`.
6	Minimize the Program Manager and arrange HELPER and CLIENT so that you can see both on the screen.
7	Now in the CLIENT application choose DDE-Initiate and click OK. ‘Select New Conversation’ will appear with a list of all DDE aware applications.
8	Click SERVER and then click OK. The DOS program will register one `WM_DDE_INITIATE` and one `WM_DDE_ACK`.
9	Choose DDE-Advise from the CLIENT menu. The ‘Advise’ dialog box will appear.
10	Type ‘Amount’ into the ‘Item’ edit control. Click the ‘Advise’ button and then ’Done’.
11	Select the DOS program and enter some data in the ‘Amount’ field and press enter. The DDE data is passed via a hot link to the CLIENT application where it can be seen in the client window.
You can enter data as many times as you wish and each time you press enter the CLIENT program is notified of the new data via DDE.	
The other choices in the DDE menu of the CLIENT application are:	
Action	**Description**
Terminate	Terminates the DDE conversation
Unadvise	Removes the hot-link established with ‘Advise’
Request	Makes a one time only request for data from the server
Poke	Pokes a value into the server field
Execute	Sends a string to the server for execution

Table 29.2 *How to use the example programs.*

The DOS Clipboard Interface

The clipboard is a piece of global memory that any number of applications can share. Applications can read from and write data to the clipboard at the user's request (i.e., whenever the user selects Edit-Cut, Edit-Copy, or Edit-Paste from a Windows menu). Under enhanced-mode Windows DOS programs can also take advantage of the clipboard. This is possible through the use of the multiplex software interrupt (`INT 2Fh`). The DOS clipboard interface is one of the few Windows facilities that Microsoft extended to DOS programs running in Windows.

From DOS, a programmer may use the following functions to transfer data to and from the clipboard:

Function	**Description**
`INT 2F` — Function `17h` Subfunction `01h`	Open clipboard
`INT 2F` — Function `17h` Subfunction `02h`	Empty clipboard
`INT 2F` — Function `17h` Subfunction `03h`	Set clipboard data
`INT 2F` — Function `17h` Subfunction `04h`	Get clipboard data size
`INT 2F` — Function `17h` Subfunction `05h`	Get clipboard data.
`INT 2F` — Function `17h` Subfunction `08h`	Close clipboard
`INT 2F` — Function `17h` Subfunction `09h`	Compact clipboard.

How They Work Together

In this case the virtual device, VDDED, actually makes direct calls into both Windows and DOS code. When you execute the Windows DDE helper application, it spawns the DOS DDE server application. Both the helper and server register their presence with the virtual device, `wddjdded.386`, installed in `system.ini`. The DOS server sends a message to the virtual device with the address of the DOS DDE call-back routine `DDECallBack`, as shown in `wddjdded.asm`(Listing 29.1). The virtual device stores this address so that it can call the DOS server whenever a DDE conversation takes place. The ability to call a DOS or Windows application

Following is an example of two functions that can be called from any DOS application to transfer data to and from the Windows Clipboard, using fixed-size (`CLIPBOARD_MAX`) global variables (`buffer_in` and `buffer_out`):

```
void CopyToClipboard()
    {
    _asm
    {
    mov ax,1701h    ; Open clipboard
    int 2fh
    mov ax,1702h    ; Empty clipboard
    int 2fh
    mov ax,1703h    ; Set clipboard data
    mov dx,01h
    mov bx,seg buffer_in ; segment of buffer
    mov es,bx
    mov bx,offset buffer_in  ; offset of buffer
    mov cx,CLIPBOARD_MAX ; size of buffer
    mov si,0
    int 2fh
    mov ax,1708h    ; Close clipboard
    int 2fh
    }
    }
```

from a virtual device is implemented in the `VMMCall SimulateFarCall` function (see Listing 29.2). On another end, the Windows DDE helper application sends a message to the virtual device with the address of its DDE callback routine. Now, when a DDE conversation begins between a client and the helper, the helper keeps the server abreast of all developments via the callback mechanism in the virtual device. Using the same mechanism, the server in turn notifies the helper of what actions to take.

Conclusion

The example described in this article does not rely upon any undocumented functions or structures. The DDE message passing is done

```
void PasteFromClipboard()
    {
    _asm
    {
    mov ax,1701h    ; Open clipboard
    int 2fh
    mov ax,1705h    ; Get clipboard data
    mov dx,01h   ; CF_TEXT
    mov bx,seg buffer_out ; segment of buffer
    mov es,bx
    mov bx,offset buffer_out ; offset of buffer
    int 2fh
    mov ax,1708h     ; Close clipboard
    int 2fh
    }
    }
```

Before allowing a user to use the clipboard, the application must check that Windows is running in enhanced mode. This can be done from DOS using the following sample function:

through the interface documented in the SDK and DDK. The method presented here of passing DDE messages to a DOS application should work in future versions of Windows. It will not, however, work in Windows NT, where the device driver model is different.

While the method used here to allow a DOS application to participate in a DDE conversation may seem convoluted, it is probably the only one available without the benefit of having access to Windows source code. There are still many DOS applications that have not been ported to Windows for one reason or another. This DOS DDE interface can help you leverage such an application in the Windows environment without incurring the cost of porting the application to Windows.

```
int IsWindowsRunningInEnhancedMode(void)
    {
    _asm
    {
    mov ax,1600h
    int 2Fh ; Is enhanced mode running?
    cmp al,0
    jne EnhancedModeIsRunning ; Yes - jump
    jmp Error      ; No - Error
    }
EnhancedModeIsRunning:
    return 1;
Error:
    return 0;
    }
```

The method presented here of passing DDE messages to an DOS program originally used the clipboard for data transfer. Now, however, all data transfer takes place directly through the control of the VxD. The reason for this change is that the clipboard is supposed to be a user-controlled Windows facility. This means that applications should put or get data from the clipboard at the user's request only. When applications use the clipboard as a private storage or transfer area, the user's data may be destroyed.

Listing 29.1 `server.c` — *DOS example server for DDE.*

```
/*****************************************************************************\
*                                                                             *
* SERVER.EXE - DOS DDE Server                                                 *
*                                                                             *
* Written by Guy Eddon, 1992                                                  *
*                                                                             *
* SERVER.EXE is WinExec by HELPER.EXE - Requires VDDED.386 and HELPER.EXE    *
*                                                                             *
* Build with MS C/C++ 7.00 or MS C 6.00 and the MS Windows 3.1 SDK:          *
*                                                                             *
*    cl /c /Gs /AL server.c                                                   *
*    link serverd,,NUL,,;                                                     *
*                                                                             *
\*****************************************************************************/

#include <WINDOWS.H>                  /* Few DOS programs ever use WINDOWS.H */
#include <DDE.H>                      /* Few DOS programs ever use DDE.H     */
#include <DOS.H>
#include <BIOS.H>
#include <CONIO.H>
#include <STDIO.H>

#define MK_FP(seg,ofs) ((void far *)(((unsigned long)(seg) << 16) | (unsigned)(ofs)))

#define REAL_MODE_CALLBACK 0x5555             /* Give VxD real mode address  */
#define UPDATE_DDE         0x7777             /* Server has data             */
#define VDDED_Dev_ID       0x305F             /* VDDED VxD ID                */
#define VID_NORMAL         0x07               /* Normal video attribute      */
#define VID_NORMAL_INT     0x0F               /* Intense video attribute     */
#define VID_REVERSE        0x70               /* Reverse vido attribute      */
#define ESC                27                 /* ESC key                     */
#define VDDED_DATA         200                /* VDDED data request          */
#define VDDED_EXECUTE      201                /* VDDED execute request       */
#define VDDED_POST         202                /* VDDED poke request          */
#define DOS_NOTIFY_QUIT    0x0299             /* Server exit notify message  */
#define DOS_COPY_TO_VXD    0x1623
#define DOS_COPY_FROM_VXD  0x1624

int read_field(unsigned char, unsigned char, unsigned char, char *);
void clear(unsigned char, unsigned char, unsigned char, unsigned char, unsigned char);
void mxyputc(unsigned char, unsigned char, char, unsigned char, unsigned char);
void mxyputs(unsigned char, unsigned char, char far *, unsigned char);
void moutchar(unsigned char, unsigned char, char, unsigned char);
void box_double(unsigned char, unsigned char, unsigned char, unsigned char);
void gotoxy(unsigned char, unsigned char);
void create_screen(void);
unsigned keyboard(void);
void CopyToVxD(void);
void CopyFromVxD(void);

void far *Apientry = 0;
char far *vid_mem_one;
char far Buffer[50] = { 0 };
```

Listing 29.1 *continued*

```
int initiate_receive,
    terminate_receive,
    advise_receive,
    unadvise_receive,
    ack_receive,
    data_receive,
    request_receive,
    poke_receive,
    execute_receive;

int initiate_send,
    terminate_send,
    advise_send,
    unadvise_send,
    ack_send,
    data_send,
    request_send,
    poke_send,
    execute_send;

void far DdeCallBack()
   {
   char string[10];
   int message, SendReceive;

   _asm mov message, ax
   _asm mov SendReceive, bx

   switch(message)
      {
      case WM_DDE_INITIATE:
         SendReceive ? initiate_receive++ : initiate_send++;
         itoa(initiate_receive, string, 10);
         mxyputs(50, 10, string, VID_NORMAL);
         itoa(initiate_send, string, 10);
         mxyputs(61, 10, string, VID_NORMAL);
         break;

      case WM_DDE_TERMINATE:
         SendReceive ? terminate_receive++ : terminate_send++;
         itoa(terminate_receive, string, 10);
         mxyputs(50, 11, string, VID_NORMAL);
         itoa(terminate_send, string, 10);
         mxyputs(61, 11, string, VID_NORMAL);
         break;
```

Listing 29.1 *continued*

```
case WM_DDE_ADVISE:
   SendReceive ? advise_receive++ : advise_send++;
   itoa(advise_receive, string, 10);
   mxyputs(50, 12, string, VID_NORMAL);
   itoa(advise_send, string, 10);
   mxyputs(61, 12, string, VID_NORMAL);
   break;

case WM_DDE_UNADVISE:
   SendReceive ? unadvise_receive++ : unadvise_send++;
   itoa(unadvise_receive, string, 10);
   mxyputs(50, 13, string, VID_NORMAL);
   itoa(unadvise_send, string, 10);
   mxyputs(61, 13, string, VID_NORMAL);
   break;

case WM_DDE_ACK:
   SendReceive ? ack_receive++ : ack_send++;
   itoa(ack_receive, string, 10);
   mxyputs(50, 14, string, VID_NORMAL);
   itoa(ack_send, string, 10);
   mxyputs(61, 14, string, VID_NORMAL);
   break;

case WM_DDE_DATA:
   SendReceive ? data_receive++ : data_send++;
   itoa(data_receive, string, 10);
   mxyputs(50, 15, string, VID_NORMAL);
   itoa(data_send, string, 10);
   mxyputs(61, 15, string, VID_NORMAL);
   break;

case WM_DDE_POKE:
   SendReceive ? poke_receive++ : poke_send++;
   itoa(poke_receive, string, 10);
   mxyputs(50, 17, string, VID_NORMAL);
   itoa(poke_send, string, 10);
   mxyputs(61, 17, string, VID_NORMAL);
   break;

case WM_DDE_EXECUTE:
   SendReceive ? execute_receive++ : execute_send++;
   itoa(execute_receive, string, 10);
   mxyputs(50, 18, string, VID_NORMAL);
   itoa(execute_send, string, 10);
   mxyputs(61, 18, string, VID_NORMAL);
   break;
```

Listing 29.1 *continued*

```
      case WM_DDE_REQUEST:
         SendReceive ? request_receive++ : request_send++;
         itoa(request_receive, string, 10);
         mxyputs(50, 16, string, VID_NORMAL);
         itoa(request_send, string, 10);
         mxyputs(61, 16, string, VID_NORMAL);
      case VDDED_DATA:            /* VDDED request data */
         CopyToVxD();
         break;

      case VDDED_EXECUTE:
         CopyFromVxD();
         mxyputc(11, 21, ' ', 50, VID_NORMAL);
         mxyputs(11, 21, Buffer, VID_NORMAL_INT);
         break;

      case VDDED_POST:
         CopyFromVxD();
         mxyputc(48, 20, ' ', 30, VID_REVERSE);
         mxyputs(48, 20, Buffer, VID_REVERSE);
         break;
      }
   }

main()
   {
   _asm
      {
      mov     ax,1600h
      int     2Fh             ; Win - Is enhanced mode running?
      cmp     al,0
      jne     EnhancedModeIsRunning   ; Yes - jump
      jmp     Error1                  ; No - Error1
```

Listing 29.1 *continued*

```
EnhancedModeIsRunning:

      mov     ax,1684h              ; Get VDDED address
      mov     bx,VDDED_Dev_ID
      int     2fh
      mov     WORD PTR Apientry,di  ; Save the callback address
      mov     WORD PTR Apientry+2,es
      mov     ax,es                 ; Is VDDED installed?
      or      ax,di
      jz      Error2                ; No - Error2
      mov     ax,1683h              ; Get current virtual machine ID in bx
      int     2fh
      mov     ax, REAL_MODE_CALLBACK
      mov     cx, seg DdeCallBack   ; segment of real mode call back
      mov     dx, offset DdeCallBack ; offset of real mode call back
      mov     si, seg Buffer        ; segment of real mode buffer
      mov     di, offset Buffer     ; offset of real mode buffer
      call    DWORD PTR Apientry    ; Call VDDED
      cmp     ax,0                  ; If Error
      jz      Error3                ; Yes - Error3
      cmp     ax,2                  ; If Error
      jz      Error4                ; Yes Error4
      jmp     OK                    ; Everything is OK
      }

Error1:
   printf("\nEnhanced Mode Windows is not running.\n");
   goto Quit;

Error2:
   printf("\nVDDED.386 is not running.\n");
   goto Quit;

Error3:
   printf("\nSERVER.EXE is already running.\n");
   goto Quit;

Error4:
   printf("\nHELPER.EXE is not running.\n");
```

Listing 29.1 *continued*

```
Quit:
   exit(0);

OK:

   create_screen();
   while(1)
      if(-1 == read_field(48, 77, 20, Buffer))
         break;
      else
         {
         _asm mov ax,VDDED_DATA
         DdeCallBack();
         _asm mov  ax,UPDATE_DDE
         _asm call DWORD PTR Apientry   ; call VDDED
         }

   clear(0, 0, 80, 25, VID_NORMAL);

   _asm mov     ax, DOS_NOTIFY_QUIT
   _asm call    DWORD PTR Apientry      ; call VDDED
   }

void CopyFromVxD(void)
   {
   _asm mov ax,DOS_COPY_FROM_VXD
   _asm call DWORD PTR Apientry   ; call VDDED
   }

void CopyToVxD(void)
   {
   _asm mov ax,DOS_COPY_TO_VXD
   _asm call DWORD PTR Apientry   ; call VDDED
   }
```

Listing 29.1 *continued*

```
void create_screen(void)
 {
  vid_mem_one = (char far *)MK_FP(0xB800, 0000);
  clear(0, 0, 80, 25, VID_NORMAL), box_double(0, 0, 79, 24);

  mxyputc(1,  2,  (char)196, 78, VID_NORMAL);
  mxyputc(0,  2,  (char)199, 1,  VID_NORMAL);
  mxyputc(79, 2,  (char)182, 1,  VID_NORMAL);

  mxyputc(1,  7,  (char)196, 78, VID_NORMAL);
  mxyputc(0,  7,  (char)199, 1,  VID_NORMAL);
  mxyputc(79, 7,  (char)182, 1,  VID_NORMAL);

  mxyputc(1,  9, (char)196, 78, VID_NORMAL);
  mxyputc(0,  9, (char)199, 1,  VID_NORMAL);
  mxyputc(79, 9, (char)182, 1,  VID_NORMAL);

  mxyputc(1,  19, (char)196, 78, VID_NORMAL);
  mxyputc(0,  19, (char)199, 1,  VID_NORMAL);
  mxyputc(79, 19, (char)182, 1,  VID_NORMAL);

  mxyputc(1,  22, (char)196, 78, VID_NORMAL);
  mxyputc(0,  22, (char)199, 1,  VID_NORMAL);
  mxyputc(79, 22, (char)182, 1,  VID_NORMAL);

  mxyputs(33, 1, "DDE DOS SERVER", VID_NORMAL);

  mxyputs(26, 3, "**  ENHANCED MODE WINDOWS  **", VID_NORMAL);
  mxyputs(26, 4, "**  SERVER.EXE HELPER.EXE  **", VID_NORMAL);
  mxyputs(26, 5, "**        VDDED.386        **", VID_NORMAL);

  mxyputs(16, 8,  "   Message                        Received    Sent ", VID_NORMAL);

  mxyputs(16, 10, "WM_DDE_INITIATE                     0          0  ", VID_NORMAL);
  mxyputs(16, 11, "WM_DDE_TERMINATE                    0          0  ", VID_NORMAL);
  mxyputs(16, 12, "WM_DDE_ADVISE                       0          0  ", VID_NORMAL);
  mxyputs(16, 13, "WM_DDE_UNADVISE                     0          0  ", VID_NORMAL);
  mxyputs(16, 14, "WM_DDE_ACK                          0          0  ", VID_NORMAL);
  mxyputs(16, 15, "WM_DDE_DATA                         0          0  ", VID_NORMAL);
  mxyputs(16, 16, "WM_DDE_REQUEST                      0          0  ", VID_NORMAL);
  mxyputs(16, 17, "WM_DDE_POKE                         0          0  ", VID_NORMAL);
  mxyputs(16, 18, "WM_DDE_EXECUTE                      0          0  ", VID_NORMAL);

  mxyputs(2,  20, "Cost of Clinton's transition team  -  Amount:", VID_NORMAL);

  mxyputs(2,  21, "Execute:", VID_NORMAL);

  mxyputs(31, 23, "Press ESC to Exit", VID_NORMAL);
  }
```

Listing 29.1 *continued*

```
void mxyputs(unsigned char x, unsigned char y, char far *str, unsigned char attr)
   {
   while(*str)
      moutchar(x++, y, *str++, attr);
   }

void mxyputc(unsigned char x, unsigned char y, char ch, unsigned char num, unsigned char
attr)
   {
   unsigned char i;
   for(i = 0; i < num; i++)
      moutchar(x++, y, ch, attr);
   }

void moutchar(unsigned char x, unsigned char y, char ch, unsigned char attr)
   {
   char far *v = vid_mem_one;
   v += y * 160 + x * 2;
   *v++ = ch, *v = attr;
   }

void box_double(unsigned char x1, unsigned char y1, unsigned char x2, unsigned char y2)
   {
   register int i;
   char far *v, far *t;
   t = v = vid_mem_one;
   for(i = y1 + 1; i < y2; i++)
      {
      v += i * 160 + x1 * 2;
      *v++ = 186;
      *v = VID_NORMAL;
      v = t;
      v += i * 160 + x2 * 2;
      *v++ = 186;
      *v = VID_NORMAL;
      v = t;
      }
   for(i = x1 + 1; i < x2; i++)
      {
      v += y1 * 160 + i * 2;
      *v++ = 205;
      *v = VID_NORMAL;
      v = t;
      v += y2 * 160 + i * 2;
      *v++ = 205;
      *v = VID_NORMAL;
      v = t;
      }
   moutchar(x1, y1, (char)201, VID_NORMAL);
   moutchar(x1, y2, (char)200, VID_NORMAL);
   moutchar(x2, y1, (char)187, VID_NORMAL);
   moutchar(x2, y2, (char)188, VID_NORMAL);
   }
```

Listing 29.1 *continued*

```
void clear(unsigned char x1, unsigned char y1, unsigned char x2, unsigned char y2,
           unsigned char attr)
   {
   register unsigned char col, row;
   for(row = y1; row < y2; row++)
      for(col = x1; col < x2; col++)
         moutchar(col, row, (char)32, attr);
   }

int read_field(unsigned char x1, unsigned char x2, unsigned char y, char *buffer)
   {
   unsigned char col = x1, row, count;
   gotoxy(x1, y);
   for(count = x1; count <= x2; count++)
      moutchar(count, y, (char)32, VID_REVERSE);
   for(count = x1; ; count++)
      {
      buffer[count - x1] = toupper(keyboard());
      if(buffer[count - x1] == '\r')
         return 0;
      if(buffer[count - x1] == ESC)
         return -1;
      if(buffer[count - x1] == '\b')
         {
         count--;
         if(col != x1)
            {
            moutchar(--col, y, (char)32, VID_REVERSE);
            count--;
            }
         }
      else
         moutchar(col++, y, buffer[count - x1], VID_REVERSE);
      if(col < x1)
         col = x1;
      else
         if(col > x2 + 1)
            {
            count--;
            col = x2 + 1;
            moutchar(col, y, (char)32, VID_NORMAL);
            }
      gotoxy(col, y);
      }
   buffer[count - x1] = 0;
   return 0;
   }
```

Listing 29.1 *continued*

```
void gotoxy(unsigned char x, unsigned char y)
   {
   union REGS i;
   i.h.dh = y;
   i.h.dl = x;
   i.h.ah = 2;
   i.h.bh = 0;
   (void)int86(16, &i, &i);
   }

unsigned far keyboard(void)
   {
   unsigned ret_val;
   while(1)
      if(_bios_keybrd(_KEYBRD_READY) != 0)
         break;
   ret_val = _bios_keybrd(_KEYBRD_READ);
   ret_val =
      (ret_val & 0x00ff) ? (ret_val & 0x00ff) : ((ret_val & 0xff00) / 0xff);
   return(ret_val);
   }

/* End of File */
```

Listing 29.2 `wddjdded.asm` — *VxD for DOS-DDE communication.*

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                            ;
; WDDJDDED.386 - DDE to DOS VxD                                              ;
;                                                                            ;
; Written by Guy Eddon, 1992                                                 ;
;                                                                            ;
; Put this in the [386Enh] section of your SYSTEM.INI file                   ;
;                                                                            ;
; Build with the MS Windows 3.1 DDK:                                         ;
;                                                                            ;
;    masm5 -p -w2 -Mx wddjdded.asm;                                          ;
;    link386 /NOI /NOD /NOP wddjdded,wddjdded.386,NUL,,wddjdded.def;         ;
;    addhdr wddjdded.386                                                     ;
;                                                                            ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

PAGE 58,132
TITLE WDDJDDED
.386p

.XLIST
INCLUDE VMM.INC
INCLUDE VDD.INC
.LIST

; #defines for communication with HERLPER and SERVER
UPDATE_DDE          equ 7777h                  ; Server has data
REAL_MODE_CALLBACK  equ 5555h                  ; Give VxD real mode address
VDDED_Dev_ID        equ 305Fh                  ; VDDED VxD ID
TEST_DOS            equ 8888h                  ; Test if Server has data
WINDOWS_CHECK       equ 8889h                  ; Helper test if Server running
EXIT_DOS            equ 8890h                  ; Server exit notify value
WINDOWS_NOTIFY_QUIT equ 8891h                  ; Helper exit notify message
CLEAR_DOS           equ 9999h                  ; Helper got Server's data
DOS_NOTIFY_QUIT     equ 0299h                  ; Server exit notify message
DOS_COPY_TO_VXD     equ 1623h                  ; Server has data
DOS_COPY_FROM_VXD   equ 1624h

WM_DDE_FIRST        equ 03E0h                  ; Same as DDE.H #defines
WM_DDE_INITIATE     equ (WM_DDE_FIRST)
WM_DDE_TERMINATE    equ (WM_DDE_FIRST+1)
WM_DDE_ADVISE       equ (WM_DDE_FIRST+2)
WM_DDE_UNADVISE     equ (WM_DDE_FIRST+3)
WM_DDE_ACK          equ (WM_DDE_FIRST+4)
WM_DDE_DATA         equ (WM_DDE_FIRST+5)
WM_DDE_REQUEST      equ (WM_DDE_FIRST+6)
WM_DDE_POKE         equ (WM_DDE_FIRST+7)
WM_DDE_EXECUTE      equ (WM_DDE_FIRST+8)
WM_DDE_LAST         equ (WM_DDE_FIRST+8)
```

Listing 29.2 *continued*

```
VDDED_CB_DATA STRUC

VDDED_State  dd   ?

VDDED_CB_DATA ENDS

Declare_Virtual_Device WDDJDDED, 3, 0, VDDED_Control, VDDED_Dev_ID, , VDDED_API_PROC,
     VDDED_API_Proc

VxD_LOCKED_DATA_SEG
ALIGN   4

WM_DDE_type                 dd  0
SendReceive                 dd  0
Pointer_To_DOS_Program      dd  0
Pointer_To_Windows_Program dd  0
DOSHasData                  dd  0
DOS_Running                 dd  0
WINDOWS_Running             dd  0
DOS_VM_ID                   dd  0
DOS_LINEAR_BUFFER           dd  0
WIN_LINEAR_BUFFER           dd  0
DOS_Seg                     dw  0
DOS_Off                     dw  0
WIN_Sel                     dw  0
WIN_Off                     dw  0

VxD_LOCKED_DATA_ENDS

VxD_ICODE_SEG

VxD_ICODE_ENDS

VxD_LOCKED_CODE_SEG

BeginProc VDDED_Control

        ret

EndProc VDDED_Control

VxD_LOCKED_CODE_ENDS

VxD_CODE_SEG
```

Listing 29.2 *continued*

```
BeginProc VDDED_API_Proc

        push    ebx                             ; Save ebx

        movzx   eax, WORD PTR [ebp.Client_BX]
        mov     SendReceive,eax                 ; SendReceive = client bx
        movzx   eax, WORD PTR [ebp.Client_AX]   ; eax = client ax switch(eax)
        cmp     eax, WINDOWS_NOTIFY_QUIT        ; case WINDOWS_NOTIFY_QUIT:
        jz      SHORT HelperIsQuiting
        cmp     eax, DOS_NOTIFY_QUIT            ; case DOS_NOTIFY_QUIT:
        jz      SHORT ServerIsQuiting
        cmp     eax, WINDOWS_CHECK              ; case WINDOWS_CHECK:
        jz      SHORT HelperCheck
        cmp     eax, CLEAR_DOS                  ; case CLEAR_DOS:
        jz      ResetVariable
        cmp     eax, TEST_DOS                   ; case TEST_DOS:
        jz      TestVariable
        cmp     eax, UPDATE_DDE                 ; case UPDATE_DDE:
        jz      Call_Windows_Program
        cmp     eax, REAL_MODE_CALLBACK         ; case REAL_MODE_CALLBACK:
        jz      SHORT real_mode
        cmp     eax, DOS_COPY_TO_VXD
        jz      get_data_from_dos
        cmp     eax, DOS_COPY_FROM_VXD
        jz      put_data_to_dos
        mov     WM_DDE_type, eax                ; default:
        jmp     Call_DOS_Program

HelperIsQuiting:
        mov     WINDOWS_Running,0               ; Helper just quit
        jmp     VDDED_exit

ServerIsQuiting:
        mov     DOS_Running,0                   ; Server just quit
        jmp     VDDED_exit

HelperCheck:
        mov     WINDOWS_Running,1               ; Helper just started
        mov     [ebp.Client_AX],1               ; Yes - return 1
        mov     ax,[ebp.Client_BX]
        mov     WIN_Sel,ax
        mov     ax,[ebp.Client_CX]
        mov     WIN_Off,ax
        jmp     VDDED_exit
```

Listing 29.2 *continued*

```
No_NOT_Running:
        mov     [ebp.Client_AX],0              ; Return 0
        jmp     VDDED_exit

real_mode:
        cmp     WINDOWS_Running,0              ; Is Helper running?
        jz      SHORT Helper_NOT_RUNNING       ; No - jump
        cmp     DOS_Running,1                  ; Yes - Is Server running
        jz      SHORT Server_ALREADY_RUNNING   ; Yes - jump
        movzx   eax,[ebp.Client_BX]            ; VM ID of DOS box
        mov     DOS_VM_ID,eax
        movzx   eax,[ebp.Client_CX]            ; No - real mode address
        mov     WORD PTR Pointer_To_DOS_Program,ax
        movzx   eax,[ebp.Client_DX]            ; Put it in Pointer_To_DOS_Program
        mov     WORD PTR Pointer_To_DOS_Program+2,ax
        mov     DOS_Running,1                  ; Return 1
        mov     ax,[ebp.Client_SI]
        mov     DOS_Seg,ax
        mov     ax,[ebp.Client_DI]
        mov     DOS_Off,ax
        mov     [ebp.Client_AX],1
        jmp     VDDED_exit

Helper_NOT_RUNNING:
        mov     [ebp.Client_AX],2              ; Return 2
        jmp     VDDED_exit

Server_ALREADY_RUNNING:
        mov     [ebp.Client_AX],0              ; Return 0
        jmp     VDDED_exit

TestVariable:
        cmp     DOS_Running,0                  ; Is Server running?
        jz      SHORT DOS_Quit                 ; No - jump
        mov     eax,DOSHasData                 ; Yes
        mov     [ebp.Client_AX],ax             ; client ax = DOSHasData
        jmp     VDDED_exit

DOS_Quit:
        mov     [ebp.Client_AX],EXIT_DOS       ; Return EXIT_DOS
        jmp     VDDED_exit

ResetVariable:
        mov     DOSHasData,0                   ; DOSHasData = 0
        jmp     VDDED_exit
```

Listing 29.2 *continued*

```
Call_DOS_Program:
        mov     eax, DOS_VM_ID                 ; Get requested VMID
        or      eax, eax                       ; non-zero?
        jz      short vmcb_ok                  ; zero, ebx is correct
vmid_scan:
        cmp     eax, [ebx.CB_VMID]             ; found it?
        je      short vmcb_ok                  ; yes
        VMMCall Get_Next_VM_Handle             ; get next
        VMMCall Test_Cur_VM_Handle             ; back to start of list?
        jne     vmid_scan                      ; no
        jmp     VDDED_exit
vmcb_ok:
        mov     esi, OFFSET32 DOSEventCallBack ; esi = &DOSEventHandler
        VMMCall Schedule_VM_Event              ; Schedule_VM_Event
        VMMCall Set_Execution_Focus            ; Switch to VM and execute call
        jmp     VDDED_exit

Call_Windows_Program:
        mov     DOSHasData,1                   ; DOSHasData = 1
        jmp     VDDED_exit

put_data_to_dos:
        mov     eax, DOS_VM_ID                 ; Get requested VMID
        or      eax, eax                       ; non-zero?
        jz      short vmcb_ok_dos1                  ; zero, ebx is correct
vmid_scan_dos1:
        cmp     eax, [ebx.CB_VMID]             ; found it?
        je      short vmcb_ok_dos1                  ; yes
        VMMCall Get_Next_VM_Handle             ; get next
        VMMCall Test_Cur_VM_Handle             ; back to start of list?
        jne     vmid_scan_dos1                      ; no
        jmp     VDDED_exit
vmcb_ok_dos1:
        mov     esi, OFFSET32 DOSGetMapFlatAddr
        VMMCall Schedule_VM_Event              ; Schedule_VM_Event
        VMMCall Set_Execution_Focus            ; Switch to VM and execute call
        VMMCall Get_Sys_VM_Handle
        mov     esi, OFFSET32 WINGetMapFlatAddr
        VMMCall Schedule_VM_Event              ; Schedule_VM_Event
        VMMCall Set_Execution_Focus            ; Switch to VM and execute call
        mov     edi,0
        mov     cx,50
        mov     eax, DOS_LINEAR_BUFFER
        mov     ebx, WIN_LINEAR_BUFFER
begin_y:
        mov     dh,byte ptr [ebx+edi]
        mov     byte ptr [eax+edi],dh
        inc     edi
        loop    begin_y
        jmp     SHORT VDDED_exit
```

Listing 29.2 *continued*

```
get_data_from_dos:
        mov     eax, DOS_VM_ID                  ; Get requested VMID
        or      eax, eax                        ; non-zero?
        jz      short vmcb_ok_dos2                   ; zero, ebx is correct
vmid_scan_dos2:
        cmp     eax, [ebx.CB_VMID]              ; found it?
        je      short vmcb_ok_dos2                   ; yes
        VMMCall Get_Next_VM_Handle              ; get next
        VMMCall Test_Cur_VM_Handle              ; back to start of list?
        jne     vmid_scan_dos2                       ; no
        jmp     short VDDED_exit
vmcb_ok_dos2:
        mov     esi, OFFSET32 DOSGetMapFlatAddr
        VMMCall Schedule_VM_Event               ; Schedule_VM_Event
        VMMCall Set_Execution_Focus             ; Switch to VM and execute call
        VMMCall Get_Sys_VM_Handle
        mov     esi, OFFSET32 WINGetMapFlatAddr
        VMMCall Schedule_VM_Event               ; Schedule_VM_Event
        VMMCall Set_Execution_Focus             ; Switch to VM and execute call
        mov     edi,0
        mov     cx,50
        mov     eax, DOS_LINEAR_BUFFER
        mov     ebx, WIN_LINEAR_BUFFER
begin_x:
        mov     dh,byte ptr [eax+edi]
        mov     byte ptr [ebx+edi],dh
        inc     edi
        loop    begin_x
        jmp     SHORT VDDED_exit

VDDED_exit:
        pop     ebx                             ; Restore ebx
        ret

EndProc VDDED_API_Proc

BeginProc DOSEventCallBack                      ; Now executing with VM focus

        Push_Client_State                       ; Save client state
        VMMCall Begin_Nest_V86_Exec             ; Enter nested code
        movzx   ecx,WORD PTR Pointer_To_DOS_Program
        movzx   edx,WORD PTR Pointer_To_DOS_Program+2
        mov     eax,WM_DDE_type                 ; eax = WM_DDE_type
        mov     ebx,SendReceive                 ; ebx = SendRecieve
        mov     [ebp.Client_AX],ax              ; client ax = ax
        mov     [ebp.Client_BX],bx              ; client bx = bx
        VMMCall Simulate_Far_Call               ; Call code in VM
        VMMCall Resume_Exec                     ; Do it
        VMMCall End_Nest_Exec                   ; Exit nested code
        Pop_Client_State                        ; Restore client state
        ret

EndProc DOSEventCallBack
```

Listing 29.2 *continued*

```
BeginProc DOSGetMapFlatAddr

        Push_Client_State                       ; Save client state
        VMMCall Begin_Nest_V86_Exec             ; Enter nested code

        mov     ax,DOS_Seg
        mov     [ebp.Client_SI],ax
        mov     ax,DOS_Off
        mov     [ebp.Client_DI],ax
        mov     ax,(Client_SI SHL 8) + Client_DI
        VMMCall Map_Flat
        mov     DOS_LINEAR_BUFFER,eax

        VMMCall Resume_Exec                     ; Do it
        VMMCall End_Nest_Exec                   ; Exit nested code
        Pop_Client_State                        ; Restore client state
        ret

EndProc DOSGetMapFlatAddr

BeginProc WINGetMapFlatAddr

        Push_Client_State                       ; Save client state
        VMMCall Begin_Nest_Exec             ; Enter nested code

        mov     ax,WIN_Sel
        mov     [ebp.Client_SI],ax
        mov     ax,WIN_Off
        mov     [ebp.Client_DI],ax
        mov     ax,(Client_SI SHL 8) + Client_DI
        VMMCall Map_Flat
        mov     WIN_LINEAR_BUFFER,eax

        VMMCall Resume_Exec                     ; Do it
        VMMCall End_Nest_Exec                   ; Exit nested code
        Pop_Client_State              ; Restore client state
        ret

EndProc WINGetMapFlatAddr

VxD_CODE_ENDS

        END                           ; That's all folks!

; End of File
```

Listing 29.3 `wddjdded.def` — *module definition file for* `wddjdded.asm`

```
LIBRARY      WDDJDDED
DESCRIPTION 'DDE to DOS VxD'
EXETYPE      DEV386
SEGMENTS
             _LTEXT PRELOAD NONDISCARDABLE
             _LDATA PRELOAD NONDISCARDABLE
             _ITEXT CLASS 'ICODE' DISCARDABLE
             _IDATA CLASS 'ICODE' DISCARDABLE
             _TEXT  CLASS 'PCODE' NONDISCARDABLE
             _DATA  CLASS 'PCODE' NONDISCARDABLE
EXPORTS
        WDDJDDED_DDB @1
```

Index

`286LOAD.ASM` *198*
`386LOAD.ASM` *219*
386MAX *115*
`@5ITDUMP.EXE` *392*
80286 `LOADALL` *189*
80386 descriptor cache entries *191*
80386 `LOADALL` data structure format *190*
8086 FPU *510*
8-bit parallel data transmission *422*

A

`Activate` *91, 94–95*
ADC *see Analog-to-Digital Converter*
`AdjustBall` *485–487, 489*
allocation chain *21–22*
`AllocSelector()` *135*
Analog-to-Digital Converter *566*
asynchronous I/O *533–534, 545*
AT bus *653*
autodependency information *393*

AUTOEXEC.BAT *53, 62, 610, 687*

B

bad clusters *18*
BALLS.INC *483–484, 487–488, 500*
bandwidth *421*
Baud Rate Divisor Latch *461–462*
BCD *see Binary-Coded Decimal*
best_rate() *569*
Binary-Coded Decimal *404*
BIOS *14*
BIOS break flag *304, 371–372, 372*
BIOS data region *115, 369*
BIOS disk services *86, 91*
BIOS functions *453*
BIOS Parameter Block *14, 47, 52, 54, 61, 691–693*
BIOS routines *52*
_bios_keybrd *85*
block device drivers *611*
boot failure *53*
boot record *14–15, 48–49, 56, 61*
boot record extensions *692*
boot sector *45–48, 52, 54, 61*
  master *55, 61*
  normal *55*
  organization *56–57*
BOOT360.ASM *47, 64–65*
BOOT4038.ASM *67–68*
BOOT.BIN *47*
BOOTPASS.ASM *69–70*
BOOTRUN *50*
BOOTRUN.ASM *71–73*
bootstrap loader
  disk *55–56*
  ROM *52, 54–55*
BOOTSYS *50*
BOOTSYS.ASM *74–75*

`BOOTSYS.BIN` *50*

Borland C/C++

  floating-point options *570–571*

Borland Turbo Debugger *392, 395*

`box()` *271*

BPB, *see BIOS Parameter Block*

BRDL, *see Baud Rate Divisor Latch*

`breakout.asm` *375*

bus master *651–652, 660*

busy bit *139*

C

C stack segment and stack pointer *87*

CACP, *see Central Arbitration Control Point*

`calibrateDelay()` *539*

`CallbackProc()` *138–139, 139*

`cancel()` *371*

`capture` *304*

`CDCNTL` *621, 628*

CD-ROM

  character device drivers *611*

  device header *611*

  extensions *610*

  locating the header *612*

  request packet *612*

CD-ROM drive *609–611, 614*

CDS, *see Current Directory Structure*

Central Arbitration Control Point *659*

chaining *95*

character device drivers *611*

`CHECK_DIR` *175*

`CheckForDebugInfo()` *396*

checking for debug information *391–402*

child programs *2*

`clean_up_for_exit()` *273*

`clear_window()` *271*

clipboard *765*

`clipbrd.exe` *132*

`clip.c` *133, 141*

`clock()` *409*

cluster *61*

cluster number *17, 21*

clusters *15–16*

  bad *18*

  reserved *18*

`cmacro.inc` *299*

`cmd.asm` *698*

`CMD.ASM` *716*

CMOS RAM *178*

code segment register *50*

CodeView

  Microsoft *391–393*

`Com_Handler()` *464*

`COMMAND.COM` *1–2, 2–3, 53, 62, 62*

`COMM.C` *462, 466*

`COMM.H` *462–463*

Common User Access, *see IBM Common User Access*

complex numbers *511*

`ComPort_Init()` *464*

`COMSPEC` *1, 3*

`CONFIG.SYS` *53, 62*

`config.sys` *116*

`CONFIG.SYS` *120, 319, 321, 598, 687, 689–690, 700, 702*

Configuration Variable Table *689–690, 699*

`CONIO.H` *83*

control word *515*

control-break and critical-error vectors *95*

converting a Microsoft C program to a TSR *77*

`copy_old_to_new()` *327*

`CPUCHECK.PAS` *183*

`CPUSTUFF.ASM` *178, 180*

`CPUSTUFF.OBJ` *178–179*

`CPU_TYPE.ASM` *257*

critical error flag *371*

critical error handling *261–300*

  abort option *266, 272*
  carry flag *268*
  device header *271*
  error descriptions *272*
  error identification *271*
  fail option *266, 272*
  stabilizing call *266*
  state of stack *262–263*
  user interface *268–269*
  user responses *272*
critical error interrupt *264*
`critical.asm` *275*
`crittest.c` *285*
`crittest.exe` *300*
`CS` register *50*
Ctrl-C and *Ctrl-Break* handling *301–317*
`ctrl_c.asm` *307*
Ctrl-C/Break flag *369*
`CtrlC_Handler()` *464*
CUA, *see IBM Common User Access*
Current Directory Structure *690–692, 699*
current stack pointer *266*
cursor control keys *358*
CVT, *see Configuration Variable Table*
cycle stealing *651*

D

DAC, *see Digital-to-Analog Converter*
data area *14–15*
Data Received Interrupt *453*
data sectors *15*
`dbginfo` *392, 396–397*
`dbginfo.c` *396*
`DBGINFO.C` *399*
`dbginfo.exe` *391*
`dd.asm` *728*
DDE, *see Dynamic Data Exchange*

DDEML, *see Dynamic Data Exchange Management Library*
`DEBUG` *47–48*
debug information checking for *391–402*
debugging DLLs *391*
debugging formats *392*
`Declare_Virtual_Device` *138*
default critical error handler, replacing *265*
`delay.asm` *546*
`delay.h` *547*
descriptor cache base registers *186*
descriptor cache contents *197*
descriptor cache entry formats *188*
descriptor cache register *188, 195–197*
descriptor table *186, 188, 196–197*
detecting a 386 processor *177–184*
determining machine stability *370*
device driver
  access to foreign segments *734*
  block driver *690–692*
  CD-ROM *609–650*
  debugging self-loading driver *700–701*
  driver chain *689–690*
  how DOS loads *688–690*
  initializing *690*
  memory structure *597*
  memory map *695*
  real mode, interfacing to Windows *727–755*
  register dump *732*
  request block *690*
  resident code *696*
  segment *612*
  self-loading *687–725*
  transfer buffer *729–731*
  TSR-like *694*
  writing with Turbo C *595-607*
device drivers,CD-ROM
  audio play *616–617*
  commands *615*

  `IOCTL READ` *617–621*
  locating the header *613–614*
  play command sequence *616*
  request packet *614–617*
  resume play *617*
device handles *323*
device header *271, 611*
`Device_TypeOfInterrupt()` *668*
`device_write()` *701*
`df_init()` *597–598*
digital vs. analog waveforms *562–563*
Digital-to-Analog Converter *563*
DIP switch *653*
Direct Memory Access, see DMA
Directory Entry Structure *20*
discardable code segments *139*
disk I/O on the PC *533*
disk organization *54*
Disk Parameter Block *699*
Disk Transfer Address *18, 95*
Disk Transfer Area *87*
`DisplayBall` *486, 489*
division by overflow method *571*
DMA, *see Direct Memory Access, 651–686*
  arbitration level *659–660*
  bus cycle *652*
  channel *653, 655–656, 659–660*
  controller *533–544, 652–665*
  controller registers *653, 655–656*
  descriptor *667*
  driver *667–669*
  enhanced mode *664*
  memory buffer *663–664, 666*
  Micro Channel controller *658–661*
  OS/2 *666*
  page register scheme *662*
  paging *664–666*
  protected mode *662–664*

- real-mode addresses *661*
- request *652*
- transfer *658, 662–665, 667–668*
- transfer address *658*
- transmit buffer *669–670*
- Windows *666*
- word transfers *657*

dmaapi.c *674*
dma.h *671*
dmaisr.c *677*
dmaPageTest() *539*
DOMOVES.ASM *184*
DOS and BIOS services, invoking *454–455*
DOS boots, modifying *45*
DOS box *759*
DOS break flag *303*
DOS drive organization *14*
DOS environment block *174*
DOS EXEC *53*
DOS File Allocation Table *13–44*
DOS file handle limits, handling *319–355*
DOS functions *453*
DOS housekeeping process *303*
DOS Load and Execute function *325*
DOS memory management *115–116*
DOS Protected Mode Interface *727–728*
DOS *Technical Reference* *266*
_dos_allocmem *81*
_dos_findfirst() *396*
_dos_findnext() *396*
_dos_freemem *81*
dos.h *410*
dosio() *599*
DOSIO.C *605*
_dos_keep *85, 87, 90*
_dos_setblock *81*
DPB, *see Drive Parameter Block*
DPL fields *189*

DPMI, *see DOS Protected Mode Interface*
  routines to access *741*
dpmiAllocNewSelector() *734*
dpmi.c *728, 733, 733–734*
dpmi.h *728*
DPMI.H *740*
dpmiReadDescriptor() *734*
dpmiRMInt() *733*
dpmiSegmentToSelector() *734*
dpmiSetDescriptor() *734*
drive organization, DOS *14*
Drive Parameter Block *21–22, 692–693*
drive sectors *14–15*
DTA, *see Disk Transfer Address*
DVR_MAIN.ASM *598*
Dynamic Data Exchange *757–786*
  links *757, 762*
  Management Library *759*
  messages *757*
dynamic swapping *118*

E

EachBall *485–486*
EMM, *see Expanded Memory Manager*
EMS
  page frame *117–118*
  driver *170*
  expanded memory *177*
ems_tsr.c *119, 127*
EMULOAD.ASM *243*
encoder Integrated Circuit *404*
End Of Interrupt *372*
enhanced keyboard *358*
ENV.ASM *3, 5–6*
environment
  global *1*
  master *2–4*

environment block *173*
environment string array *174*
environment variables *1–3*
EOI, *see End Of Interrupt*
EQUATES.INC *483, 499*
error handling, critical *261–300*
errs.h *296*
event timing *583–594*
EXE2BIN *47, 50*
exec_command() *698, 702*
expanded memory, loading TSRs into *115–130*
Expanded Memory Manager *116, 118*
Expanded Memory Specification *116, 118*
  handle *119*
  mapping context *119*
  page frame *119–120*
exploiting the 386 four-byte moves *177–184*
extended memory *177*
EXtended Memory Specification *116*
  device driver *116*
  himem.sys *116*
EXTND16.ASM *361*

F

FAT, *see File Allocation Table*
FAT mechanics *15*
FATWALK *14, 21–22*
FATWALK.ASM *23–44*
FCB *18*
FCBs, *see File Control Blocks*
FDC, *see Floppy Disk Controller*
fdcDelay() *539*
fdcReadSectors() *542–543*
fdcRecalibrate() *542, 542*
fdcReset() *541*
fdcResetParms() *540*
fdcSeek() *542*

fdcSendCMD() 541
fdcSenseDrive() 541
fdcSenseInterrupt() 542
fdcSetDriveParms() 541
fdcSetDrivParms() 540
fdcWaitForInt() 545
fdcWriteSectors() 543
File Allocation Table *13,44*
File Control Block Structure *19*
File Control Blocks *173–174*
file fragmentation *14*
file handle information blocks *321*
file transfer with parallel port *421–460*
files, hidden *18*
Find First Matching File With FCB 18
find_pfht() 326
floating point math *509–510*
Floating Point Unit *510*
Floppy Disk Controller
  commands *536, 541–543*
  communicating with *534*
  data register *536, 541*
  debugging FDC programs *545*
  Digital Output Register *534–535*
  initializing *540–541*
  interfacing *533–559*
  Main Status Register *535*
  numbering *537–538*
  parameters *540*
  status registers *543–544*
  terminology *537*
  timing *536*
floppy.c 548
floppy.h 557
flush_kbd_buffer() 272
FORMAT 21
Forth-style stack trace *512*
FPSUBS.ASM 511–512, 512, 515–516

`fpsubs.asm` 524
FPU, *see Floating Point Unit*
  emulating a coprocessor 515
  exceptions 516
  memory synchronization 514
  stack 512
FPU stack 512

G

game port
  accessing 405
  AT BIOS routines 412
  connector type 407
  pinouts 407
  programming 403
GDT and IDT descriptor cache entry formats 189
`generate_codes()` 570
`Get Default Drive Parameter Block` 21
`Get DOS Version` 266
get key 304, 306, 359
`Get List of Lists` 165
`get shift status` 359
`get_critical_error_addr()` 371
`Get_Data()` 669
`get_indos_addr()` 370
`GetPSP` 94
`GetShift` 91
global environment block 2–3
global MS-DOS environment 1–12
`GlobalEnvDel` 3
`GlobalEnvDelD` 4
`GlobalEnvGet` 3
`GlobalEnvPut` 3–4
`GlobalEnvSeg` 3, 3
`GlobalEnvSize` 3, 3
`GLOBENV.C` 3–4
`globenv.c` 7–11

`GLOBENV.H` *3*
`globenv.h` *12*
gracefully breaking out of infinite loops *369–390*

H

half-duplex port *423*
handling DOS file handle limits *319–355*
`HasCodeViewStyleInfo()` *396*
`HasTDStyleInfo()` *396*
hidden files *18*
hidden register *186, 195*
hidden windows *757*
High Memory Area *115*
High Sierra addressing mode *616*
HMA, *see High Memory Area*
hot key *78, 78, 83–85, 87, 91, 91, 94, 119, 370–373*
hot key shift mask *87*
huge drive *21–22*

I

`i08hndlr()` *586*
`I1CEX` *588*
`i8swapout()` *586*
IBM Common User Access *133*
`IBMBIO.COM` *52*
IC, *see encoder Integrated Circuit*
ICE, *see In Circuit Emulator*
ICE trace *191*
ICE trace of 386 `LOADALL` *193–194*
IER, *see Interrupt Enable Register*
IIR, *see Interrupt Identification Register*
important ROM BIOS interrupts *59*
In Circuit Emulator *186*
increasing DOS file handle limits *320*
`inc_sfht.c` *351*
`InDosPtr` *95*
infinite loops, gracefully breaking out of *369*

initblk.asm 699–700, 720
initialization of data areas 51
init_new_pfht() 327
InitPSP 94
inportb() 409
INS 8250 chip 457, 463
INS 8250 registers 458–462
install24() 265
Install_Handlers() 464
install_vectors 304
instruction pointer register 50
insure() 371–372
INT 21h AH=52h 165
int11.c 582
int24() 265–266
int86() 412
int_8h_handler() 372
interfacing to the floppy disk controller 533–559
interfacing Windows to a real-mode device driver 727–755
interrupt() 696–699, 701
Interrupt Enable Register 453, 459, 463–464
interrupt handler 452
Interrupt Identification Register 459–460, 464
Interrupt Vector Port 452
Interrupt Vector Table 262, 265, 453–454, 463, 465, 584
interrupt-driven serial communications 451–477
interrupt-driven serial I/O 462
IntHandler() 135
inverted bus-style pins 428
IOCTL READ 617–621
IO.SYS 50, 52, 56, 62
IP register 50
IPX/SPX Event Service Routines 119
ISA 653
IterateThroughFiles() 396

J

joystick *403–407*
  button presses *409–410*
  button status *412*
  calibration *411–412*
  functions *411*
  timing loop *409*
joystick ports *403*
joystick timing loop *408*
`joytest.c` *411, 415*
jump table *699*

K

key code *358–360*
keyboard buffer *302, 306, 358–360*
keyboard handlers *93*
keyboard vector *95*

L

LCR, *see Line Control Register*
`LH`, *see also* `LOADHIGH`, 117
LIM/EMS, *see Expanded Memory Specification*
Line Control Register *460*
Line Status Register *458*
List of Lists *165, 322–323, 688–690*
`LOADALL` *177, 185–260*
  80286 *189*
  ICE trace of 386 `LOADALL` *193–194*
`LOADALL` emulation *191*
`loadhi.c` *118–120, 122–126*
`LOADHIGH` *117*
Local Descriptor Table *734*
logical disks *54*
logical pages *116*
LOL, *see List Of Lists*

lost keystrokes, preventing *357–367*
LSR, *see Line Status Register*

M

machine state *370*
Making Windows and DOS programs talk *131–162*
mantissa *510*
MASM *178*
master boot sector *54–55, 61*
master environment *2–4*
math coprocessor *509–532*
maximum cluster number *21, 21*
max_sys_file_handles() *323–324*
MCB, *see Memory Control Block*
MCR, *see Modem Control Register*
media ID byte *18*
memory
  non-volatile *50*
  volatile *50*
memory chain
  MS-DOS *164*
  walking the *165*
Memory Control Block *3, 116–119, 164–165*
memory image *186*
memory organization *58*
memory resident program, see Terminate-and-Stay-Resident
Micro Channel bus *653–659*
Micro Channel DMA controller *658–661*
Microsoft CD-ROM extensions *610*
Microsoft CodeView *185, 391–392, 394*
MIDI, *see Musical Instrument Digital Interface*
minimizing size of resident stub *173*
Modem Control Register *461, 464–465*
Modem Status Register *461*
modifying DOS boots *45–75*
monitor() *269*
MS-DOS environment, global *1–12*

MS-DOS memory chain *164*
MS-DOS timing chain *584–585*
`MSDOS.SYS` *52–53, 56, 62*
MSR, *see Modem Status Register*
Musical Instrument Digital Interface *404*

N

`NCLUS` *22*
`NewInt09` *90–91, 91*
`NewInt13` *90*
`NewInt23` *91*
`NewInt28` *91, 91, 95*
`NewInt9` *95*
`NEWT.C` *511*
`newt.c` *518*
Newton's method *511*
`NEXT_CLUSTER` *22*
non-inverting pins *428*
non-volatile memory *50*
NPX, *see Numeric Processor eXtension*
Numeric Processor Extension *191*

O

OCR output circuit *424*
OEMs *138*
`oldexe.h` *395–396*
`OLDEXE.H` *398*
one shot R/C timing network *405*
`OpenFile()` *730*
original interrupt *370*

P

page frame, EMS *117–118*
`paracopy.c` *432, 435*
`paracopy.h` *432, 449*
parallel port

  base address *428*
  cable wiring scheme *432*
  description of *423–426*
  electrical characteristics *426, 428*
  file transfer protocol *430*
  for high-speed file transfer *421–450*
  interface description *427*
  interrupt *434*
  null cable *431*
  other alternative uses *433*
  polling sequence *431*
  port access *428*
  sending and receiving bytes *429*
  status and control pins *429*
  test harness *433*
parameters for 3.5-inch floppies *66*
`paraport.c` *433, 439*
parent and child programs *2*
partition *61*
partition table *54–56, 62*
pass-key check *48*
`PATH` *1–4*
PC BIOS interface *533*
PC speaker
  floating-point options *570–571*
  generating input streams *571*
  generating instruction streams *570*
  generating the audio *566–567*
  mechanical integrator *564–565*
  pulse-width modulation *565*
  sending bits to the speaker *567–568*
  writing for *561–582*
Periscope's debugger *303*
`pfht.asm` *335*
PIC, *see Programmable Interrupt Controller*
`PIPE_AllocateSelector()` *139*
`pipe.asm` *138, 140, 149*
`PIPE_Call_Windows_Procedure()` *139*

pipe.def 139, 148
pipedos.c 138–139, 154
PIPE_Event_Procedure() 139
pipefunc.c 138, 159
PIPE_RegisterCallback() 138
PIPE_UnregisterCallback() 139
pipewin.c 138–139, 156
pipewin.def 139, 155
PIT, *see Programmable Interval Timer*
playing CD audio discs *609*
pm.c 728
pm.exe 728, 730, 734
pmiCallDD() 729
POOL.ASM 483, 485–487, 489, 491
POST, *see Power-on Self Test*
PostMessage() 139
potentiometers *404*
Power-on Self Test 51, 62
PreActivate 91
preemptive multitasking *760*
preventing lost keystrokes *357–367*
Process File Handle Table *325–328*
process file handles *322*
process_handles_free() 325
Program Segment Prefix *2, 87, 93–95, 134, 167, 170–171, 173, 265, 322, 324–325, 372, 696*
Programmable Interrupt Controller *453, 464, 585*
Programmable Interval Timer *585–586*
programming the PC game port *403*
PROMPT 1–2
PSP, *see Program Segment Prefix*
pulse-width modulation *565*
Put_Data() 669

Q

QEMM-386 *115*
QuickC development environment *93*

R

R/C *405*
R/C network *407–408*
RDR, *see Receiver Data Ready*
Read_Char() *464*
real-mode memory, reclaiming for other uses *115*
receive side ring buffer *599*
Receiver Data Ready *459*
Red Book addressing mode *616, 619–620*
registers, hidden *186*
release *306*
relocate.asm *339*
relocate_pfht() *326–327*
replacing error handlers *265*
reserved clusters *18*
reserved sectors *54*
resident stub, minimizing size of *173*
resistor/capacitor timing network *405, 405*
restore_window() *273*
reverse engineering *391*
rm.c *728*
ROM, bootstrap loader *52, 54–55*
ROM BIOS *52*
ROM BIOS interrupts *59*
root directory *14–15, 20, 54*
routines to access DPMI *741*
RQH.H *602*
RS-232 port *452*
RS-232 port *462*
RS-232 serial interface standard *422*

S

save_window() *269*
sayat() *271, 271*
scan code *358–359*
SCR, *see Scratch Register*

Scratch Register 461
screen flicker 485
scrnstuf.asm 287
ScrRestoreBlock 81
ScrSaveBlock 81
sectors 14
segment descriptor caches 186
segment overrides 189
segment register 195
self-loading device drivers 687–725
serial communications, interrupt-driven 451–477
serial port 452–453
  interrupts 453
  programming 462–463
server.c 763, 769
SET 2, 4
setcw() 515
SetPSP 94
SETREG 483–484
SetSelectorBase() 135
SetSelectorLimit() 135
sfht.asm 329
shared memory 137
SHOW_CLUSTER 22
SimulateFarCall 767
sixteen_handler 304
sldd.asm 705
sldd.inc 694, 704
software-visible segment registers 188
spawn() family, swapping replacement for 163–176
SPAWNENV.C 174
SPAWNO 163–176
SPAWNPTH.ASM 175
__SPAWN_SEARCH 175
__SPAWNV.ASM 174
spawnvp() 119
special entry code 373
square() 516

starting cluster number *21*
`start_up` *697–699*
status word *512, 515–516*
`strategy()` *599, 696, 698–699, 701*
`stub_tsr.c` *119, 128–129*
swap file *168*
swapping replacement for the `spawn()` family *163–176*
`SYS.COM` *62*
`sys_file_handles_in_use()` *323–324*
`SYSINIT` *52, 52–53*
SysReq *358*
System File Handle Table *321–322, 325*
system file handles *320*
System VM *759, 761*

T

`tctimer.c` *327*
`tctimer.h` *327*
`tdrv.c` *701, 723*
`TDUMP` *392*
Terminate-and-Stay-Resident *2, 172, 357, 434, 587, 621, 694, 700, 730, 735*
  activating *86–87*
  and QuickC *83*
  API services to other programs *120*
  converting to *77–114*
  emulating 286 `LOADALL` with 386 `LOADALL` *192*
  error codes *80*
  for preventing lost keystrokes *357–367*
  floating point operations *82*
  guidelines *81–82*
  how TSR functions work *85*
  initialization *78*
  interfacing DOS with Windows *134–135*
  interrupt chaining *120*
  loading into expanded memory *115–130*
  must not attempt child process *82*
  removing itself from memory *84*

  shared memory *137–138*
`test16.c` *304, 315*
`testbrk.c` *389*
`TESTCALL.ASM` *183*
`testpfht.c` *338*
`testrelo.c` *344*
`testsfht.c` *334*
`test_time()` *569*
`testtime.c` *348*
`TEXT` *3*
`_TEXT` *696*
the 386 four-byte moves, exploiting *177*
`thirteen_handler()` *370*
THR, *see Transmitter Holding Register*
`time.h` *409*
`TOSPKR` *566, 568–572*
`tospkr.c` *573*
transient application *81*
transmit buffer *599*
transmit side ring buffer *599*
Transmitter Holding Register *458*
trapping *Ctrl-C* and *Ctrl-Break* *304*
trapping DOS critical errors *261–300*
TSR *see Terminate-and-Stay-Resident programs*
`TSRA.ASM` *86, 105–106*
`TSR.C` *81, 96–104*
`tsr.c` *134, 143*
`TSRDEMO.C` *108–114*
`TSR.H` *78, 84, 107*
`TsrInDos` *78, 83–84, 87, 95*
`TsrInstall` *78, 80, 84, 87, 87, 91, 94*
`TsrRemove` *78, 84–85, 95*
`tsrwin.c` *134–135, 145*
`tsrwin.def` *134, 144*
TTL logic *421*
TTL output circuit *423*
Turbo Debugger *392, 395*
`twenty_five_handler()` *370*

`twenty_six_handler()` *370*

U

`u2bin.c` *580*

UART, *see Universal Asynchronous Receiver and Transmitter*

UMBs, *see Upper Memory Blocks*

Universal Asynchronous Receiver and Transmitter *457*

`update_pfht_addr()` *327*

`update_pfht_count()` *327*

Upper Memory Blocks *115–116, 118*

user interrupt vector *95*

`user_interface()` *269*

`user_io.c` *280*

`UserRtn` *87*

using CD-ROM device drivers *609–650*

V

VDD, *see Virtual Display Device*

VDDED *766*

VDS, *see Virtual DMA Services*

VDS Lock DMA Region service *666*

`VERIFY_CLUSTER` *22*

vertical sync pulse *482*

VGA *(Virtual Graphics Adapter)*
  - color planes *480–481, 489–490*
  - determining scan line *486*
  - erasing image *486*
  - modes *479*
  - page flipping *481, 485*
  - palette *480*
  - scan lines *480, 482, 490*
  - screen *480*
  - screen flicker *482, 485*
  - vertical sync pulse *482, 485*
  - video display memory *480–481, 483–484, 488–490*
  - video display mode *483*

VGA animation graphics *479–508*

video display memory *480–481, 484, 488–490*
video display mode *483*
virtual 8086 machine *730*
Virtual Device Driver *135–139, 757, 761*
Virtual Display Device *761*
Virtual DMA Services *666–667*
Virtual Machine Manager *759*
VM *134, 730*
VM architecture *136*
VMM, *see Virtual Machine Manager*
volatile memory *50*
VxD, *see Virtual Device Driver*

W

walking the memory chain *165*
`wddjdded.386` *761–763, 766*
`wddjdded.asm` *763, 766, 779*
`wddjdded.def` *763*
`WindowProc()` *139*
Windows
  clipboard *132–133*
  enhanced mode *131–132*
  talking to DOS *131–162*
Windows DPMI client-server architecture *136*
Windows system architecture *760*
`WINOLDAP` *133*
`Winproc_CallbackCS` *138*
`WinProc_CallbackIP` *138*
`Write_Char()` *464*
`write_other_blocks` *171*
writing device drivers with Turbo C *595–607*
writing for the PC speaker *561–582*

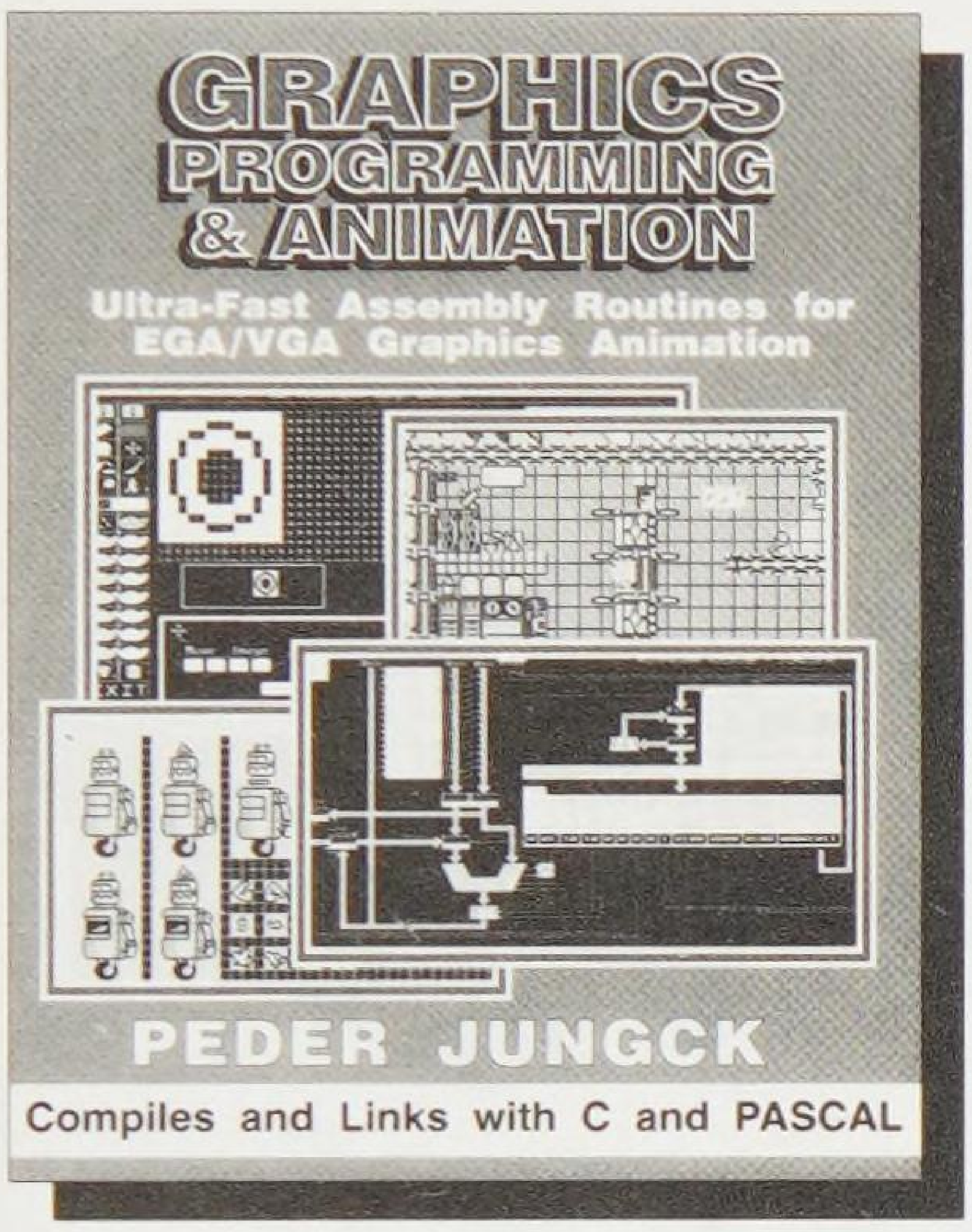
X

XMS, *see extended Memory Specification*
XMS driver *169*

Design Ultra-Fast Graphics Animation!

Graphics Programming & Animation

by Peder Jungck


GRAPHICS PROGRAMMING & ANIMATION
Ultra-Fast Assembly Routines for EGA/VGA Graphics Animation
PEDER JUNGCK
Compiles and Links with C and PASCAL

Learn how to control the video board with very fast assembly routines for EGA/VGA graphics animation. Jungck shows you how to manipulate the hardware directly and match algorithms to hardware processes, efficiently and effectively. You'll also explore the secrets of Jungck's *ProGraphx Toolbox* (used to develop best-selling shareware video games such as *Secret Agent*). Primitive functions are coded in assembly language for speed. Higher level functions are provided in both C and Pascal.

Manage PCX Graphics Easily with Flexible Data Structures

Discover how you can use modular design and put display control information in data structures for easy changes and expansions. Using the techniques of Jungck's *ProGraphx Toolbox,* you can manage screen controls with matrices, linked lists, and tables, instead of relying on specifically coded functions.

You'll also learn how to read and display image files using the PCX format, how to convert Windows™ icons into ProGraphx images, and how to edit images using a mouse-driven icon editor.

Book with Disk
$49.95
Book only - $29.95
Plus shipping

The companion disk contains over 170 program files from the book, including the complete *ProGraphx Toolbox* and a **FREE** copy of ***Secret Agent***.

Order Your Copy Today!

Use code **T86C** for ***Graphics Programming & Animation*** with disk.
Use code **T86** to order book only.

913-841-1631
FAX: 913-841-2624

R&D
Technical Books

VISA MasterCard

OPERATING SYSTEMS: DOS

DISK INCLUDED DO NOT DEMAGNETIZE

MS-DOS®
system programming

This trove of 29 tricks has something for every DOS programmer. You'll learn about:

- Writing for the PC Speaker.
- Navigating the File Allocation Table.
- Programming the Game Port (for more than games).

The enclosed disk includes ready-to-use routines worth weeks of time and cyber-sweat. See how to:

- Build device drivers that load directly from the command line.
- Break gracefully out of infinite loops.
- Interface a Windows program with real-mode device driver.
- Play music through a CD-ROM device driver.

You'll discover things you may not know about DOS and things you may know but have never had the chance to explore. You'll find complete source code that will:

- Execute high-speed file transfers using the eight-bit bandwidth of the parallel port.
- Swap all but 500 bytes of the calling program out of memory and provide a drop-in replacement to the *spawn...()* family functions.
- Disable CTRL-C and eliminate the annoying ^C from the video display.
- Check the number of available file handles without opening a single file or device and increase the number of available file handles without modifying *CONF.SYS*.

Whether you're a beginner or a veteran, you'll find *something* in this collection that will save you time and lead you to more efficient MS-DOS systems programming.

David Burki is a programmer/analyst employed by PDA, Inc., working with MS-DOS, Windows, and OS/2. His most recent development projects have been an e-mail system with a GUI interface for a local network of PCs running OS/2 and a database processor for targeted marketing using demographic profiles.

Robert Ward is a well-known technical author and lecturer. He wrote the book *Debugging C* and founded R&D Publications, Inc.

R&D Technical Books

DISK INCLUDED

ISBN 0-13-207382-X

90000>

EAN

9 780132 073820